An Adaptive Bit-Level Text Compression Scheme Based on the HCDC Algorithm

اسلوب تكيُّفي لضغط البيانات النصية بمستوى (Bit) بالاعتماد على الخوارزمية (HCDC)

By

Ahmad Rababa'a

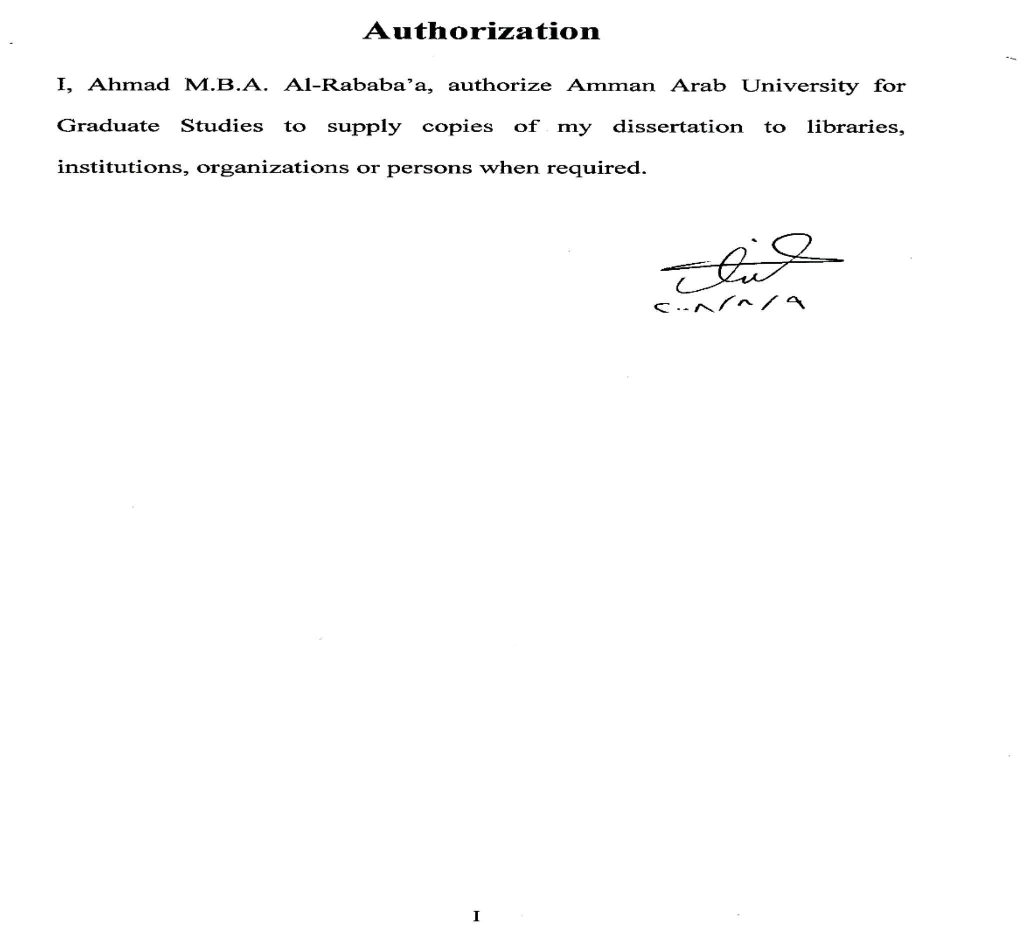
Supervisor

Dr. Hussein Al-Bahadili

This thesis is submitted to the Department of Computer Science, Graduate College of Computing Studies, Amman Arab University for Graduate Studies in partial fulfilment of the requirement for the degree of Master of Science in Computer Science.

Graduate College of Computing Studies
Amman Arab University for Graduate Studies
(Jul, 2008)

Authorization



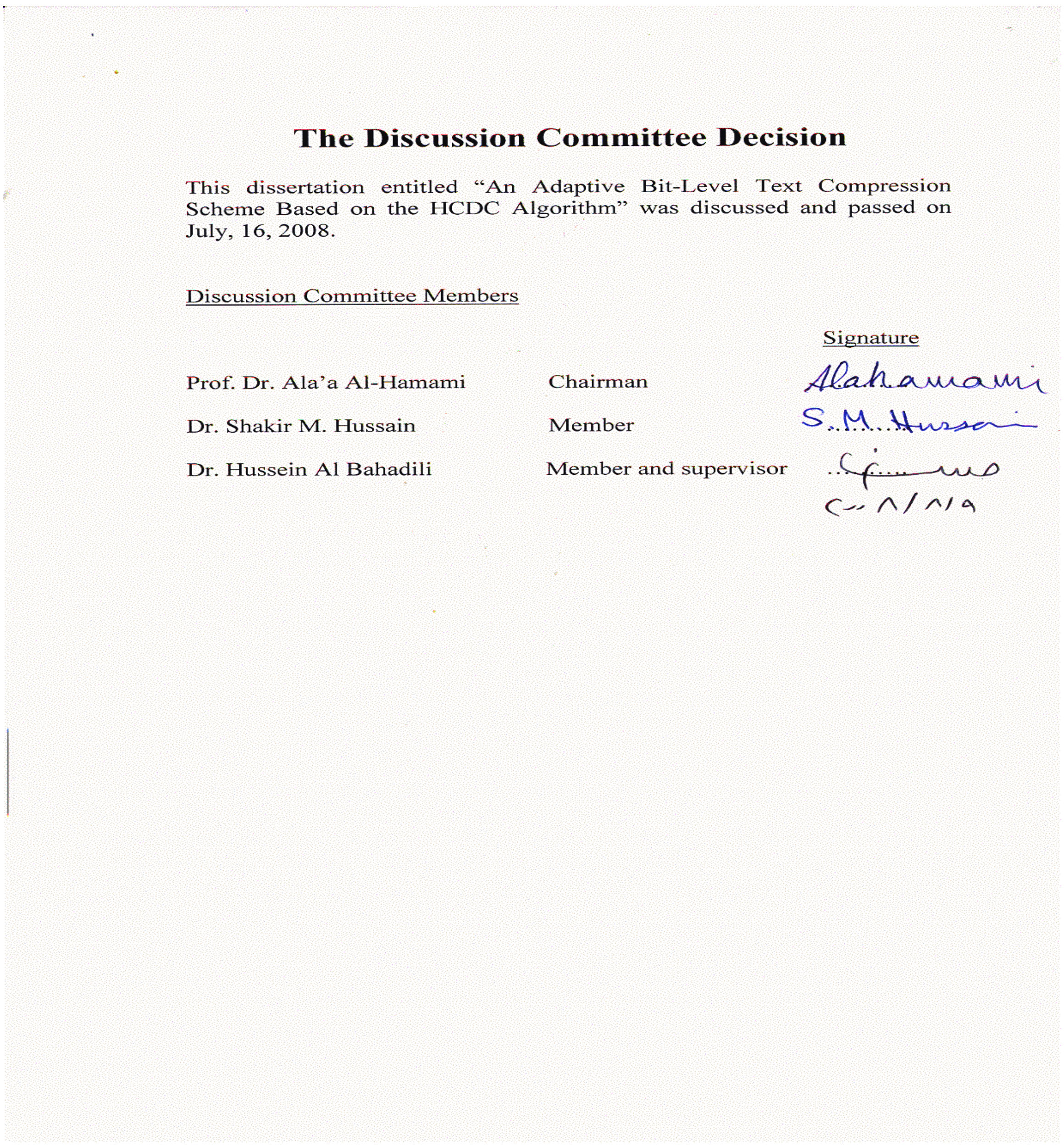
Authorization

I, Ahmad M.B.A. Al-Rababa'a, authorize Amman Arab University for Graduate Studies to supply copies of my dissertation to libraries, institutions, organizations or persons when required.

[illegible]

٢٠٠٨/٨/٩

I



The Discussion Committee Decision

This dissertation entitled “An Adaptive Bit-Level Text Compression Scheme Based on the HCDC Algorithm” was discussed and passed on July, 16, 2008.

Discussion Committee Members

		Signature
Prof. Dr. Ala’a Al-Hamami	Chairman	Alahamami
Dr. Shakir M. Hussain	Member	S.M. Hussain
Dr. Hussein Al Bahadili	Member and supervisor	[illegible] [illegible]

Abstract

This thesis is concerned with the development and performance evaluation of a new adaptive bit-level text compression scheme that is based on the Hamming Codes Data Compression (HCDC) algorithm. The HCDC algorithm is a lossless binary (bit-level) data compression algorithm that utilizes the well-known error correcting Hamming codes.

The new scheme consists of six steps some of which are applied repetitively to enhance the compression ratio. The repetition loops continue until inflation is detected. The overall (accumulated) compression ratio is the multiplication of the compression ratios of the individual loops, therefore we refer to this new scheme as HCDC(*k*), where *k* refers to the number of repetition loops.

In the HCDC(*k*) scheme, a new adaptive text-to-binary coding format was developed and used. This method of coding reduces the entropy of the generated binary sequence so that it grants higher compression ratio.

The HCDC(*k*) scheme was implemented in C++ programming language; and used to compress a number of text files from standard corpora. The results obtained demonstrate that the HCDC(*k*) scheme has higher compression ratio than most well-known text compression algorithms, and also exhibits a competitive performance with respect to many widely-used state-of-the-art software. Finally, the results obtained are discussed, conclusions are drawn, and recommendations for future work are pointed-out.

ARABIC SUMMARY

الملخَّص

أن هذة الاطروحة معنيَّة بتطوير وتقييم اداء خطة تكيُّفية جديدة لضغط البيانات النصية بمستوى (Bit) والتي تعتمد على الخوارزمية المسماة (HCDC). الخوارزمية (HCDC) هي خوارزمية لضغط البيانات بمستوى (Bit) في النظام الثنائي دون خسارة للبيانات والتي تستخدم ترميز (Hamming) لتصحيح الاخطاء. هذة الخطة الجديدة تتكون من ستة خطوات، البعض منها تطبق بشكل متكرر لتحسين نسبة الضغط ، يستمرهذا التكرار لحين تحسس التضخم. تكون نسبة الضغط الاجمالية (التراكمية) هي حاصل ضرب نسب الضغط في كل عملية تكرار منفردة، لذلك يتم الاشارة لهذة الخطة بأسم HCDC(*k*) حيث *k* تدل على عدد مرات التكرار.

في الخطة HCDC(*k*) تم تطوير صيغة تكيُّفية جديدة لتشفير النص لبيانات ثنائية ، هذة الطريقة تقلل من ال (Entropy) للتسلسل الثنائي المتولد وبالتالي يمنح نسبة ضغط اعلى.

تم برمجة الخطة HCDC(*k*) بأستخدام لغة البرمجة C++ وأستُخدمت البرمجية لضغط عدد من الملفات النصية من مجموعة Corpora القياسية.

تبيِّن النتائج أن هذة الخطة توفر نسبة ضغط اعلى من معظم خوارزميات ضغط النص المعروفة وتعرض أداء منافس مقارنة مع الكثير من البرمجيات الواسعة الانتشار. اخيرا نوقشت النتائج وتم الخروج بالنتائج وذكربعض التوصيات للعمل المستقبلي.

Acknowledgment

I would like to specially thank my supervisor Dr. Hussein Al-Bahadili, who taught me every thing that I know about research and the way it should be done. I would like to thank him for his guidance during all stages of this research, for answering endless questions, for his great support, professional advice, and profound understanding. Through this work, Dr. Hussein has shown me how to attack a problem from different angels, how to approach it, and how to find the most suitable solution.

I also would like to thank all members of staff at Amman Arab University for Graduate Studies, in particular, the members of staff at the Graduate College of Computing Studies.

Finally, it would be unthinkable of me not to thank my parents, wife, children, brothers and friends for their support and encouragement over the years. I am thankful for anyone who supported me during my study towards the Master.

Table of Contents

List of Tables

List of Figures

Abbreviations

ACW	Adaptive Character Word length
AF	Adaptive Fano
AH	Adaptive Huffman
ASCII	American Standard Code For Information Interchange
BOA	Bayesian Optimization Algorithm
BPC	Bit Per Character
BWT	Burrows–Wheeler transform
CHT	Condensed Huffman Table
CPU	Central Processing Unit
DETEC	Dynamic End Tagged Dense Code
DLETDC	Dynamic Lightweight End Tagged Dense Code
EC	Escape Code
FGK	Faller ,Gallager and Knuth algorithm

FLH	Fixed Length Hamming
Gzip	Gun Zone Improvement Plan
HCDC	Hamming Code Data Compression
HTML	Hyper Text Markup Language
HU	Huffman Coding
HF	HU following FLU
LDPC	Low- Density Paraty Check
LZ	Lempel-Ziv
LZB	Lempel-Ziv Bell
LZJ	Lempel-Ziv Jakobsson
LZMW	Lempel-Ziv Miller and Wegman
LZR	Lempel-Ziv Rodeh
LZSS	Lempel-Ziv Storer and Syzmanski
LZT	Lempel-Ziv Tischer
LZW	Lempel-Ziv-Welch
MB	Mega Byte

MTF	Move – to- Front
PBE	Byte Pair Encoding
PNG	Portable Network Graphics
RAM	Random Access Memory
PPM	Prediction with Partial Matching
RLE	Run length Encoding
SCC	Special Compression Character
SCM	Structure Context Model
SCMHuff	Structure Context Model Huffman
XML	Extensible Mark up Language

Chapter One
Introduction

1.1. Definition of Data Compression

Data compression aims to reduce the size of data so that it requires less disk space for storage and less bandwidth to be transmitted over data communication channels [Sal 04]. Data compression also reduces the amount of errors during data transmission over error-prune data communication channels by decreasing the size of information to be exchanged over such channels [Adi 07, Fre 04]. An additional benefit of data compression is in wireless communication devices where it may introduce a significant power saving. Power saving is possible by compressing data prior to transmission power consumption, where the power consumed is directly proportional to the size of the transmitted data. In fact, in wireless devices, transmission of a single bit can require over 10^3 times more power than a single 32-bit computation [Bar 06].

Data compression is usually obtained by substituting a shorter symbol for an original symbol in the source data, containing the same information but with a smaller representation in length. The symbols may be characters, words, phrases, or any other unit that may be stored in a dictionary of symbols and processed by a computing system [Wit 04].

Data compression requires efficient algorithmic transformations of a source message to produce representations (also called codewords) that are more compact. Such algorithms are known as data compression algorithms or data encoding algorithms. Each data compression algorithm needs to be complemented by its inverse, which is known as a data decompression algorithm (or data decoding algorithm), to restore an exact or an approximate form of the original data.

Data coding techniques have been widely used in developing data compression algorithms, since coding techniques may lend themselves well to the above concept. Data coding involves processing an input sequence [Rue 06]:

$$\boldsymbol{X} = \{x[1],\ x[2], \ldots x[M]\} \qquad (1.1)$$

Where each input symbol, x[i], is drawn from a source alphabet:

$$\boldsymbol{S} = \{s_i,\ s_i\ \ldots,\ s_m\} \qquad (1.2)$$

Whose probabilities are:

$\square = \{\square_1, \square_2, . . ., \square_m\}$ with $2 \leq m < \infty$ (1.3)

For example, for binary alphabet m=2, ***S*** is either 0 or 1.

The encoding process is rendered by transforming X into an output sequence:

$\boldsymbol{Y} = \{y[1], y[2], \ldots, y[Q]\}$ (1.4)

Where each output symbol y[i] is drawn from a code alphabet:

$\boldsymbol{A} = \{a_1, a_2\ldots, a_q\}$ (1.5)

Here the code alphabet is still binary (i.e., either 0 or 1, q=2), but Q must be much less than M. The main problem in data compression is to find an encoding scheme that minimizes the size of Y, in such a way that X can be completely recovered by applying the decompression process.

An algorithm or coding function is called distinct if its mapping from source messages to codewords is one-to-one. Such a code is called uniquely decodable if every codeword is recognizable even when immersed in a stream of other codewords.

A uniquely decodable code is known as a prefix code if it has the property that no codeword in the code is a prefix of any other codeword.

1.2. Categorization of Data Compression Algorithms

Data compression algorithms are categorized by several characteristics, such as:

i. Data compression fidelity
ii. Length of data compression symbols
iii. Data compression symbol table
iv. Data compression cost

Following is a brief definition for each of them.

1.2.1 Data compression fidelity

One of the most important characteristics is the fidelity with which the original and the decompressed data agree with each other. The decompressed (restored) data can either represent an exact or an approximate form of the original data set [Sha 06].

Therefore, two fundamentally different styles of data compression can be recognized, depending on the fidelity of the restored data, these are: (i) lossless data compression, and (ii) lossy data compression.

i. Lossless data compression

It involves a transformation of the representation of the original data set in such a way that it is possible to reproduce exactly the original data (exact copy). Lossless compression is used in compressing text files, executable codes, word processing files, database files, tabulation files, and whenever it is important that the original and the decompressed files must be identical.

Lossless compression is used in many applications, for example, the popular ZIP file format and in the UNIX tool gzip. It is also used as a component within lossy data compression technologies. Lossless compression algorithms can usually achieve a 2 to 8 compression ratio [Rue 06, Bri 07].

ii. Lossy data compression

It involves a transformation of the representation of the original data set in such a way that it is impossible to reproduce exactly the original data set, but an approximate representation is reproduced by performing a decompression transformation. This type of compression is used frequently on the Internet and especially in streaming media and telephony applications. Because some information is discarded, it achieves better data compression ratios that reach 100 to 200, depending on the type of information being compressed. In addition, higher compression ratio can be achieved if more errors are allowed to be introduced into the original data [Bri 07, Wit 04].

1.2.2 Length of data compression symbols

Data compression algorithms are characterized by the length of the symbols an algorithm processes, regardless of whether the algorithm uses variable length symbols in the original data or in the compressed data, or both. For example, Run-Length Encoding (RLE) uses fixed length symbols in both the original and the compressed data. Huffman encoding uses variable length compressed symbols to represent fixed-length original symbols. Other methods compress variable-length original symbols into fixed-length or variable-length encoded data.

1.2.3 Data compression symbol tables

Another distinguishing feature of the data compression algorithms is the source of the symbol table. According to this feature, data compression algorithms can be classified into:

i. Static or fixed data compression algorithms

ii. Dynamic or adaptive data compression algorithms

iii. Semi adaptive data compression algorithms

i. Static or fixed data compression algorithms

Some data compression algorithms operate on a static symbol table, or a fixed dictionary of compression symbols. Because the dictionary is fixed, it needs not be combined with the compressed data. Such algorithms are dependent on the format and content of the data to be compressed. However, a fixed dictionary is usually optimized for a particular data type, whereas if the same dictionary used for other types of information the efficiency of the algorithm suffers and provides a lower compression ratio.

ii. Dynamic or adaptive data compression algorithms

Some data compression algorithms are relatively independent, and some make two passes at the data. The first pass determines the frequency of the symbols that will be processed; and builds a symbol table based on that frequency. The custom symbol table needs to combine the compressed data, and the second pass uses the custom symbol table to encode and decode data.

Adaptive compression algorithms build a custom symbol table as they compress the data. Such algorithms encode each character based on the frequency of preceding characters in the original data file. The decompression algorithm builds an identical dynamic table as the information is decompressed. Adaptive methods usually start with a minimal symbol table to bias the compression algorithm toward the type of data they are expecting.

iii. Semi adaptive data compression algorithmes

In a semi-adaptive algorithm the data to be compressed are first analyzed in their entirety, an appropriate model is then built, afterwards the data is encoded. The model is stored as part of the encoded data, as it is required by the decompressor to reverse the encoding.

Static schemes are similar to this, but a representative selection of data is used to build a fixed model, which is hard-coded into compressors and decompressors.

This has the advantage that no model must be explicitly stored with the compressed data, but the disadvantage is that poor compression will result if the model is not representative of data presented for compression.

Concerning the type of adaptively being adopted, focus has remained on static and semi-adaptive techniques, and little attention has been paid to the class of adaptive algorithms [Kle 00, Xie 03, Gil 06].

1.2.4 Data compression cost

The cost of data compression is an important feature that can be used to distinguish between the different data compression algorithms. Most importantly is that compression algorithms should be performed in as minimum as possible cost. This cost is measured in terms of time and storage requirement; however, in many applications, and with the revolutionary advancement in computer technology, the time is the most important factor. For example, on-the-fly compression algorithms, such as between application programs and storage devices, the algorithm should operate as quickly as the storage devices themselves.

Likewise, if a compression algorithm is built into a hardware data communications component, the algorithm should not prevent the full bandwidth of the communication media from being continuously utilized.

The data compression cost for a particular algorithm consists of the time required by the algorithm to compress the original data and the time it takes to decompress the data back to its original form. In the context of the data compression for minimal storage applications, the cost of compression can be viewed as a one-time cost and hence as relatively less significant than the cost of decompression, which must be incurred every time the data is to be retrieved from storage. In the context of compression designed for fast data transmission applications, the relative costs of compression at one end and decompression at the other may be equally significant.

According to the compression-decompression processing time, data compression algorithms are classified into two classes, these are:

i. Symmetric data compression algorithms

In a symmetric data compression algorithm, the processing times are almost the same for both compression and decompression processes.

ii. Asymmetric data compression algorithms

In an asymmetric data compression algorithm the compression processing time is more than the decompression processing time.

1.3. Data Compression Models

Different data compression algorithms have been recommended and used throughout the years. These data compression algorithms can be classified into four major models; these are [Pan 00]:

i. Substitution data compression model.

ii. Statistical data compression model.

iii. Dictionary-based data compression model.

iv. Bit-level data compression model.

A substitution data compression model involves the swapping of repeating characters by a shorter representation. Algorithms that are based on this model include: Null Suppression, Run-Length Encoding (RLE), Bit Mapping and Half Byte Packing [Smi 97, Pan 00].

A statistical data compression model involves the generation of the shortest average code length based on an estimated probability of the characters. Examples of algorithms that are based on this model include: Shannon-Fano coding [Rue 06, Rue 04], static/dynamic Huffman coding [Huf 52, Knu 85,Vit 89,Vit 87], and arithmetic coding [How 94, Wit 87].

A dictionary-based data compression model involves the substitution of substrings by indices or pointer code, relative to a dictionary of the substrings; algorithms that can be classified as a dictionary-based model include the LZ compression technique and its variations [Ziv 77, Ziv 78, Nel 89, Bri 07].

Finally, since data files could be represented in binary digits, a bit-level processing can be performed to reduce the size of data. In bit-level data compression algorithms, the binary sequence is usually divided into groups of bits that are called minterms, blocks, sub sequences, etc. These minterms might be considered as representing a Boolean function.

Then, algebraic simplifications are performed on these Boolean functions to reduce the size or the number of minterms, and hence, the number of bits representing the output (compressed) data is reduced as well. Examples of such algorithms include: the Adaptive

Character Word length (ACW(*n*)) algorithm [Bah 08b].

The Adaptive Character Word length (ACW(*n,s*)) scheme [Bah 08b, Hay 08], the logic-function simplification algorithm [Nof 07], the neural network based algorithm [Mah 00].

1.4. Text Compression

1.4.1 Syllables and words based text compression

There are a number of data compression techniques that have been developed throughout the years. Some of which are of general use, i.e., can be used to compress files of different types (e.g., text files, image files, video files, etc.). Others are developed to efficiently compress a particular type of files. In this work, we are concerned with text files compression.

Text compression can often derive its benefit from the two views of the textual content: the content can be seen as a stream of syllables or words. The word-based methods are older, so many implementations of classical methods exist, for instance Huffman coding [Wit 94, Huf 52], LZW [Dvo 99], Burrows-Wheeler transformation [Isa 01], PPM [Adi 06] or Arithmetic coding [Mof 98]. The syllable-based methods are rather young with initial implementations of Huffman coding and LZW [Lan 05].

Porting of classical character-based methods to syllable or word based is not easy. The transformation heavily influences almost all inner data structures, because they must be able to work with undefined number of syllables or words instead of the original alphabet of 256 characters. Moreover, the large input alphabet also requests the encoder to export elements of the alphabet to the decoder. This issue is often solved by exporting the alphabet as a part of the encoded document [Lan 06a].

The confrontation and comparison of the word and syllable based methods depends on a language of the input document. Accordingly, the languages with a simple morphology, e.g. English, are better compressed by the word-based algorithms. On the other hand, the languages with a complex morphology are often compressed better by the syllable-based methods.

1.4.2 Bit-level text compression

Text files compression can also be performed at bit-level, as each character has its specific binary representation. However, bit-level data compression algorithms are even younger than the syllable and word based data compression algorithms, therefore, there are only few algorithms that have been developed to exploit this concept.

Recently, a lossless binary (bit-level) data compression algorithm that is based on the error correcting Hamming codes, namely, the HCDC algorithm was proposed and its performance was analyzed analytically [Bah 08a]. In the HCDC algorithm, the binary sequence to be compressed is divided into blocks of *n* bits length. To utilize the concept of Hamming codes, the block is considered as a Hamming codeword that consists of *p* parity bits and d data bits (*n*=*d*+*p*). Then each block is tested to find if it is a valid or a non-valid Hamming codeword. For a valid block, only the d data bits preceded by 0 are written to the compressed file, while for a non-valid block all *n* bits preceded by 1 are written to the compressed file. These additional 0 and 1 bits are used to distinguish the valid and the non-valid blocks during the decompression process.

This thesis is concerned with the development and performance evaluation of a new adaptive bit-level text compression scheme that is based on the HCDC algorithm. The new scheme consists of six steps some of which are repetitively applied to achieve higher compression ratio. The repetition loops continue until inflation is detected. The overall compression ratio is a multiplication of the compression ratios of the individual loops; therefore we refer to the new scheme as HCDC(*k*), where *k* refers to the number of repetition loops.

In order to enhance the compression power of the HCDC(*k*) scheme, a new adaptive encoding format is proposed in which a character is encoded to binary according to its probability of occurrence. This method of encoding reduces the binary sequence entropy so that it grants higher compression ratio. The scheme is implemented in C++ programming language and used to compress a number of text files from standard corpora.

1.5. Performance Evaluation Parameters

In order to be able to compare the efficiency of the different compression techniques reliably, and not allowing extreme cases to cloud or bias the technique unfairly, certain issues need to be considered. The most important issues need to be taken into account in evaluating the performance of various algorithms includes the following [Yia 06]:

i. Measuring the amount of compression.
ii. Processing time (Algorithm complexity).

These issues need to be carefully considered in the context for which the compression algorithm is used. Practically, things like finite memory, error control, type of data, and compression style (adaptive/dynamic, semi-adaptive or static) are all factors that should be considered in comparing the different data compression algorithms [Bel 90].

1.5.1 Measuring the amount of compression

Several parameters are used to measure the amount of compression that can be achieved by a particular data compression algorithm, such as:

i. Compression ratio (C).
ii. Reduction ratio (R).
iii. Coding rate (C_r)

Following is a brief definition for each of them.

i. Compression ratio (C)

The amount of compression is measured by a factor known as compression ratio (C), which is defined as the ratio between the size of the data before compression and the size of the data after compression. It is expressed as:

$$C = \frac{S_o}{S_c} \qquad (1.6)$$

Where S_o and S_c are the sizes of the original and the compressed data, respectively.

ii. Reduction ratio (*R*)

The reduction ratio represents the ratio between the difference between the size of the original data (S_o) and the size of the compressed data (S_c) to the size of the original data, which is referred to as R. It is usually given in percents and it is mathematically expressed as:

$$R = \frac{S_o - S_c}{S_o} \times 100 \qquad (1.7)$$

iii. Coding rate (*C_r*)

The coding rate expresses the same concept at the compression ratio, but it relates the ratio to a more tangible quantity. For example, for a text file, the coding rate may be expressed in "bits/character"

(bpc), where in uncompressed text file a coding rate of 7 or 8 bpc is used. In addition, the coding rate of an audio stream may be expressed in "bits/analogue", and for still image compression, the coding rate is expressed by "bits/pixel". The coding rate is expressed as:

$$C_r = \frac{q \cdot S_c}{S_o} \tag{1.8}$$

Where *q* is the number of bit representing each symbol in the uncompressed file. The relationship between the coding rate (*C_r*) and the compression ratio (*C*), for example, for text file originally using 7 bpc, can be given by:

$$C_r = \frac{7}{C} \tag{1.9}$$

It is clear from Eqn. (1.9) that a lower coding rate indicates a higher compression ratio.

1.5.2 Processing time

The processing time (which is an indication of the algorithm complexity) is defined, as the time required compressing or decompressing the data. These compression and decompression

times have to be evaluated separately. As it has been discussed in section 1.3, data compression algorithms are classified according to the processing time into either symmetric or asymmetric algorithms. For a symmetric algorithm, both the compression and the decompression processing time are almost the same, while for an asymmetric algorithm, usually, the compression time is much more than the decompression time.

In this context, data storage applications are mainly concerned with the amount of compression that can be achieved and the decompression processing time that is required to retrieve the data back (asymmetric algorithms).

Data transmission applications focus predominately on reducing the amount of data to be transmitted over communication channels, and both compression and decompression processing times are the same at the respective junctions (symmetric algorithms) [Liu 05].

For a fair comparison between the different available algorithms, it is important to consider both the amount of compression and the processing time. Therefore, it would be useful to be able to parameterize the algorithm in such a way that the compression ratio and processing time could be optimized for a particular application.

There are extreme cases where data compression works very well or in other conditions where it is inefficient, the type of data that the original data file contains and the upper limits of the processing time have an appreciable effect on the efficiency of the technique selected. Therefore, it is important to select the most appropriate technique for a particular data profile in terms of both data compression and processing time [Pan 00, Rue 04].

1.6. Statement of the Problem

The statement of the problem can be summarized as follows:

i. Develop an efficient bit-level text compression scheme that is based on the HCDC algorithm.
ii. Develop a suitable binary-to-character coding format.
iii. Investigate the effect of the coding format on the entropy of the generated binary sequence and how does it affect the performance of the new scheme.
iv. Evaluate the performance of the new scheme and compare its performance with other standard data compression algorithms and state-of-the-art software.

v. Discuss the results obtained, draw conclusions and point-out some recommendations for future work.

1.7. Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 presents a literature review that summarizes the most recent and related work. It is presented in two sections, one reviews the most recent work that is related to the early data compression algorithms (e.g., substitution, statistical, and dictionary-based algorithms). The other section reviews the bit-level data compression algorithms.

Chapter 3 provides a detailed description of the HCDC(*k*) scheme. The HCDC algorithm and the new adaptive coding format are also explained in this chapter. Chapter 4 presents a number of experiments that are performed to evaluate, compare, and discuss the compression ratios achieved by the new scheme. First, the compression ratios achieved by the new scheme over a number of text files from standard corpora using different coding formats are compared.

Then, the performance of the HCDC(*k*) scheme with adaptive coding is evaluated and compared with many widely used compression algorithms and state-of-the-art software. In Chapter 5, conclusions are drawn and recommendations for future work are pointed-out.

Finally, in Appendix A an introduction to the standard data compression corpora (e.g., Calgary, Canterbury, Artificial, and Large and Miscellaneous corpora), which are widely used in comparing the performance of the different data compression algorithms, are presented.

Chapter Two
Literature Review

A large number of data compression algorithms have been developed and used throughout the years. Some of which are of general use, i.e., can be used to compress files of different types (e.g., text files, image files, video files, etc.). Others are developed to compress efficiently a particular type of files. It has been realized that, according to the representation form of the data at which the compression process is performed, text compression algorithms can be broadly classified into two main classes, these are:

i. Bit-level text compression algorithms

ii. Syllable or word based text compression algorithms.

This thesis is mainly concerned with the development and performance evaluation of a bit-level text compression scheme, namely the HCDC(*k*) scheme, therefore Section 2.1 provides a review on the most recent bit-level text compression algorithms

. In Section 2.2., syllable or word based text compression algorithms are reviewed.

2.1.Bit-Level Text Compression Algorithms

H. Al-Bahadili [Bah 08a] developed a lossless binary data compression scheme that is based on the error correcting Hamming codes. It was referred to as the HCDC algorithm. In this algorithm, the binary sequence to be compressed is divided into blocks of *n* bits length. To utilize the Hamming codes, the block is considered as a Hamming codeword that consists of *p* parity bits and d data bits (*n*=*d*+*p*).

Then each block is tested to find if it is a valid or a non-valid Hamming codeword. For a valid block, only the d data bits preceded by 1 are written to the compressed file, while for a non-valid block all n bits preceded by 0 are written to the compressed file. These additional 1 and 0 bits are used to distinguish the valid and the non-valid blocks during the decompression process.

An analytical formula is derived for computing the compression ratio as a function of block size, and fraction of valid data blocks in the sequence. The performance of the HCDC algorithm was analyzed, and the results obtained were presented in tables and graphs. H. Al-

Bahadili concluded that the maximum compression ratio that can be achieved by this algorithm is $n/(d+1)$, if all blocks are valid Hamming codewords.

H. Al-Bahadili and H. Shakir [Bah 08b] proposed a bit-level data compression algorithm, in which the binary sequence is divided into blocks each of n-bit length. This gives each block a possible decimal values between 0 to 2^n-1. If the number of the different decimal values (d) is equal to or less than 256, then the binary sequence can be compressed using the n-bit character wordlength. Thus, a compression ratio of approximately $n/8$ can be achieved. They referred to this algorithm as the Adaptive Character Wordlength (ACW) algorithm, since the compression ratio of the algorithm is a function of n, it was referred to it as the ACW(n) algorithm.

Implementation of the ACW(n) algorithm highlights a number of issues that may degrade its performance, and need to be carefully resolved, such as: (i) If d is greater than 256, then the binary sequence can not be compressed using n-bit character wordlength, (ii) the probability of being able to compress a binary sequence using n-bit character wordlength is inversely proportional to n, and (iii) finding the optimum value of *n* that provides maximum compression ratio is a time consuming process, especially for large binary sequences. In addition, for text compression, converting text to binary using the equivalent ASCII code of the characters gives a high entropy binary sequence, thus only a small compression ratio or sometimes no compression can be achieved.

W. Al-Hayek [Hay 08] developed an efficient implementation scheme to enhance the performance of the ACW(*n*) algorithm, and overcome all the drawbacks mentioned above. In this scheme the binary sequence was divided into a number of subsequences (*s*), each of them satisfies the condition that *d* is less than 256, therefore it is referred to as the ACW(*n*,*s*) scheme. The scheme achieved compression ratios of more than 2 on most text files from most widely used corpora.

S. Nofal [Nof 07] proposed a bit-level files compression algorithm. In this algorithm, the binary sequence is divided into a set of groups of bits, which are considered as minterms representing Boolean functions. Applying algebraic simplifications on these functions reduce in turn the number of minterms, and hence, the number of bits of the file is reduced as well. To make decompression possible one should solve the problem of dropped Boolean variables in the simplified functions. He investigated one possible solution and their evaluation shows that future work should find out other solutions to render this technique useful, as the maximum possible compression ratio they achieved was not more than 10%.

A. Jaradat et al. [Jar 06] proposed a file splitting technique for the reduction of the nth-order entropy of text files. The technique is based on mapping the original text file into a non-ASCII binary file using a new codeword assignment method and then the resulting binary file is split into several sub files each contains one or more bits from each codeword of the mapped binary file. The statistical properties of the sub files are studied and it was found that they reflect the statistical properties of the original text file which was not the case when the ASCII code is used as a mapper.

The nth-order entropy of these sub files was determined and it was found that the sum of their entropies was less than that of the original text file for the same values of extensions. These interesting statistical properties of the resulting subfiles can be used to achieve better compression ratios when conventional compression techniques were applied to these sub files individually and on a bit-wise basis rather than on character-wise basis.

K. Barr and K. Asanovi'c [Bar 06] presented a study of the energy savings possible by lossless compressing data prior to transmission. Because Wireless transmission of a single bit can require over 1000 times more energy than a single 32-bit computation. It can therefore be beneficial to perform additional computation to reduce the number of bits transmitted.

If the energy required to compress data is less than the energy required to send it, there is a net energy savings and an increase in battery life for portable computers. This work demonstrated that, with several typical compression algorithms, there was actually a net energy increase when compression was applied before transmission. Reasons for this increase were explained and suggestions were made to avoid it. One such energy-aware suggestion was asymmetric compression, the use of one compression algorithm on the transmit side and a different algorithm for the receive path. By choosing the lowest-energy compressor and decompressor on the test platform, overall energy to send and receive data can be reduced by 11% compared with a well-chosen symmetric pair, or up to 57% over the default symmetric scheme.

The value of this research is not merely to show that one can optimize a given algorithm to achieve a certain reduction in energy, but to show that the choice of how and whether to compress is not obvious. It is dependent on hardware factors such as relative energy of CPU, memory, and network, as well as software factors including compression ratio and memory access patterns. These factors can change, so techniques for lossless compression prior to transmission/reception of data must be re-evaluated with each new generation of hardware and software.

Caire et. al [Cai 04] presented a new approach to universal noiseless compression based on error correcting codes. The scheme was based on the concatenation of the Burrows-Wheeler block sorting transform (BWT) with the syndrome former of a Low-Density Parity-Check (LDPC) code.

Their scheme has linear encoding and decoding times and uses a new closed-loop iterative doping algorithm that works in conjunction with belief-propagation decoding.

Unlike the leading data compression methods, their method is resilient against errors, and lends itself to joint source-channel encoding/decoding; furthermore their method offers very competitive data compression performance.

A. A. Sharieh [Sha 04] introduced a Fixed-Length Hamming (FLH) algorithm as enhancement to Huffman Coding (HU) to compress text and multimedia files. He investigated and tested these algorithms on different text and multimedia files. His results indicated that the FLH following HU and HU following FLH enhance the compression ratio.

A. Jardat and M. Irshid [Jar 01] proposed a very simple and efficient binary run-length compression technique. The technique is based on mapping the non-binary information source into an equivalent binary source using a new fixed-length code instead of the ASCII code. The codes are chosen in such a way that the probability of one of the two binary symbols; say zero, at the output of the mapper is made as small as possible. Moreover, the "all ones" code is excluded from the code assignments table to ensure the presence of at least one "zero" in each of the output codewords.

Compression is achieved by encoding the number of "ones" between two consecutive "zeros" using either a fixed-length code or a variable-length code. When applying this simple encoding technique to English text files, they achieve a compression of 5.44 bpc and 4.6 bpc for the fixed-length code and the variable length (Huffman) code, respectively.

M. V. Mahoney [Mah 00] introduced a fast text compression with neural network model that produces better compression than popular Limpel-Ziv compressors (zip, gzip, compress), and is competitive in time, space, and compression ratio with PPM and Burrows-Wheeler algorithms.

The compressor, a bit-level predictive arithmetic encoder using a 2-layer, 4×10^6 by 1 network, is fast (about 10^4 characters/second) because only 4-5 connections are simultaneously active and because it uses a variable learning rate optimized for one-pass training. He showed that it is practical to use neural networks for text compression in any application that requires high speed.

2.2. Syllable and Word Based Text Compression Algorithms

J. Adiego et. al [Adi 07] described a compression model for semi-structured documents, called Structural Contexts Model (SCM), which

takes advantage of the context information usually implicit in the structure of the text. The idea is to use a separate model to compress the text that lies inside each different structure type (different XML tag). The intuition behind SCM was that the distribution of all the texts that belong to a given structure type should be similar, and different from that of other structure types.

They mainly focused on semi-static models, and tested their idea using a word-based Huffman method. This was a standard for compressing large natural language text databases, because random access, partial decompression, and direct search of the compressed collection were possible. This variant is dubbed as SCMHuff, and it retained those features and improved Huffman's compression ratios.

L. Galambos et. al [Gal 07] discussed the selection of a suitable compression method, which would utilize the semantics, and structure of HTML documents. Their guess was that such a method has the best chance to achieve an optimal level of a compression. Three branches of compression algorithms were discussed: textual, special XML, and a mix of the previous two. Last branch was represented by a XBW algorithm, which combines textual method with an XML compression method.

E. Conley and S. Klein [Con 06] introduced the notion of multilingual-text compression.

The basis of multilingual-text compression is first the ability to match the corresponding parts of related texts by identifying semantic correspondences across the various sub-texts, a task generally referred to as text alignment. Some methods for detailed alignment use an existing multilingual glossary, but all of them generate their own probabilistic glossary, which corresponds to the processed text. The idea is to save storage space by replacing words and phrases with pointers to their translations, determined by any alignment algorithm. Unaligned words are compressed on their own using HuffWord encoding. The suggested method was tested on an English-French corpus of the European Union. They obtained a compression ratio of 22%, which is similar to the performances of Bzip and HuffWord and better than that of Gzip.

S. Rein et. al [Rei 06a] proposed a lossless compression method for short data series (larger than 50 Bytes). The method uses arithmetic coding and context modeling with a low-complexity data model. A data model that takes 32 KBytes of RAM already cuts the data size in half. The compression method just takes a few pages of source code, is scalable in memory size, and may be useful in sensor or cellular networks to spare bandwidth. S. Rein et. al demonstrated that their method allows for battery savings when applied to mobile phones. Further work on very short text files compression can be found in [Rei 06b, Lan 06b].

L. Robert and R. Nadarajan [Rob 06] developed a few algorithms for random access text compression in which there is a direct access to the compressed data, so that it is possible to start decompression from any place in the compressed file. If any byte changed during transmission, the remaining data can be retrieved safely. Their work was based on the Byte Pair Encoding (BPE) Scheme. The BPE algorithm was based on the fact that ASCII character set uses only codes from 0 through 127.

That frees up codes from 128 through 255 for use as pair codes. Pair code is a byte, used to replace the most frequently appearing pair of bytes in the text file. Five algorithms are developed based on this BPE scheme. These algorithms find the unused bytes at each level and try to use those bytes for replacing the most frequently used bytes. These algorithms compress typical text files approximately half of their original size, but of course, the actual amount of compression depends on the data being compressed. In these algorithms, most of the time is spent on searching for the most frequently occurring pairs. However, decompression is very fast in all these algorithms.

N. Brisaboa , et. al [Bri 05] addressed the problem of adaptive compression of natural language text, focusing on the case where low bandwidth is available and the receiver has little processing power, as in mobile applications. Their technique achieves compression ratios around 32% and requires very little effort from the receiver. This tradeoff, not previously achieved with alternative techniques, is obtained by breaking the usual symmetry between sender and receiver dominant in statistical adaptive compression.

Moreover, they showed that their technique could be adapted to avoid decompression at all cases where the receiver only wants to detect the presence of some keywords in the document.

This is useful in scenarios such as selective dissemination of information, news clipping, alert systems, text categorization, and clustering. The asymmetry they introduced, enable the receiver to search the compressed text much faster than the plain text. This was previously achieved only in semi-static compression scenarios. They improved the existing results on word-based adaptive compression, focusing on reducing the effort of the receiver in order to either uncompress or search the compressed text.

A. Moffat et. al [Mof 05] enhanced the performance of the block-sorting algorithm, which is an innovative compression mechanism introduced by Burrows and Wheeler. It involves three steps: permuting the input one block at a time using the Burrows–Wheeler transform (BWT), applying a move-to-front (MTF) transform to each of the permuted blocks, and then entropy coding the output with a Huffman or arithmetic coder. Block-sorting implementations have assumed that the input message is a sequence of characters. They extended the block-sorting mechanism to word-based models. They also considered other transformations, and were able to show improved compression results compared to MTF and uniform arithmetic coding. For large files of text, the combination of word-based modeling, BWT, and MTF-like transformations allowed excellent compression effectiveness to be attained within reasonable resource costs.

I. Witten [Wit 04] showed that the text mining is about inferring structure from sequences representing natural language text, and may be defined as the process of analyzing text to extract information that is useful for particular purposes. Although handcrafted heuristics are a common practical approach for extracting information from text, a general, approach requires adaptive techniques.

He studied the way in which the adaptive techniques to use a text compression approach in text mining. He developed several examples: extraction of hierarchical phrase structures from text, identification of key phrases in documents, locating proper names and quantities of interest in a piece of text, text categorization, word segmentation, acronym extraction, and structure recognition. He concluded that compression forms a sound unifying principle that allows many text-mining problems to be tacked adaptively.

R. Hashemian [Has 03] presented a new Huffman coding and decoding technique in which there is no need to construct a full size Huffman table in this technique

; instead, the symbols were encoded directly from the table of code-lengths. For decoding purposes a new Condensed Huffman Table (CHT) was also introduced. It was shown

that by employing this technique both encoding and decoding operations became significantly faster, and the memory consumption became much smaller compared to the normal Huffman coding/decoding.

A. Chu [Chu 02] presented a new universal lossless data compression algorithm derived from the popular and widely used LZ77 family. He referred to it as LZAC. The objective of LZAC was to improve the compression ratios of the LZ77 family while retaining the family's key characteristics: simple, universal, fast in decoding, and economical in memory consumption. LZAC presented two new ideas: composite fixed-variable-length coding and offset difference coding. A composite fixed-variable-length coding combined fixed-length coding and variable-length coding into a single coding scheme.

Rueda et al. [Rue 01] present an enhanced version of the static Fano method, namely Fano+. They formally analyzed Fano+ by presenting some properties of Fano trees, and the theory of list rearrangements. The enhanced algorithm achieved compression ratios arbitrarily close to those of Huffman's algorithm. Empirical results on files of the Canterbury corpus corroborate the almost-optimal efficiency of the enhanced algorithm and its canonical nature

H. Plantinga [Pla 06] proposed a heuristic for text compression via diagram replacement and a fast entropy coding method. The resulting compression algorithm is an asymmetric algorithm, in the sense that compression requires much time and memory, but decompression is fast and requires little memory. The algorithm is also classified as a semi-adaptive, since it allows a random access into the compressed file without decompressing the whole file.

Compression ratios achieved are competitive with gzip for a standard corpus of texts, and better for large files. The algorithm is well suited to applications for which expensive compression of large files is acceptable but decompression must be inexpensive, and high compression ratios or random access into the compressed file are required.

Chapter Three
The Adaptive Bit-Level Text Compression Scheme

This chapter provides a detail description of an adaptive, lossless, symmetric, bit-level, text compression scheme, which is based on the Hamming Codes Data Compression (HCDC) algorithm [Bah 08a]. The scheme consists of six steps some of which are repetitively applied to achieve higher compression ratio. The repetition loops continue until inflation is detected. The overall compression ratio is the multiplication of the compression ratios of the individual loops, therefore we refer to the new scheme as HCDC(*k*), where *k* refers to the number of repetition loops.

In order to enhance the compression power of the HCDC(*k*) scheme, a new adaptive coding format is used in which characters are encoded to binary according to their frequency of occurrence. In contrast to ASCII code, this method of encoding reduces the entropy of the generated binary sequence so that it grants higher compression ratio.

Section 3.1 provides an introduction to the entropy of English text and how it affects the compression ratio and the coding rate that can be achieved by a particular data compression algorithm over the text.

Different coding formats can be used in converting text into binary sequence, such as the standard ASCII code, Huffman codes, adaptive codes, etc. In Section 3.2 a description is given for the adaptive coding format and how it can be used to convert a sequence of characters into a binary sequence.

The concept of bit-level data compression is introduced in Section 3.3. Section 3.4 presents a detailed description of the HCDC algorithm. This section also presents the derivation and analysis of the compression ratio of the HCDC algorithm. The HCDC(*k*) scheme is described in Section 3.5. Finally, in Section 3.6, the components of the compressed file header of the HCDC(*k*) scheme are defined.

3.1. Entropy of English Text

The entropy is a statistical parameter that can be used to measure how much information is produced on the average for each letter of a text in the language. If the language is translated into binary digits (0 and 1) in the most efficient way, the entropy *E*, is the average number of binary digits required per letter of the original language. The

redundancy, on the other hand, measures the amount of constraint imposed on text in the language due to its statistical structure [Sha 51].

For a set of possible messages *M*, the entropy is defined as:

$$E(M) = \sum_{m \in M} p(m) \cdot i(m) \qquad (3.1)$$

where *p(m)* is the probability of message *m*. *i*(*m*) is the notion of the self information of a message and it is given by:

$$i(m) = \log_2\left(\frac{1}{p(m)}\right) \qquad (3.2)$$

This self-information represents the number of bits of information contained in it and, roughly speaking, the number of bits that should be used to represent that message. Larger entropies represent more information, and perhaps counter-intuitively, the more random a set of messages (the more even the probabilities) the more information they contain on average.

The amount of information that is contained by a text could be used as a bound to the maximum amount of compression that can be achieved.

As it has been mentioned in Chapter 1, one way to measure the information content is in terms of the average number of bits per character (bpc), i.e., coding rate (C_r). Table (3.1) shows a few approaches that can be used to measure the amount of information contained by an English text in bpc.

If all characters are assumed to have equal probabilities, a separate code is used for each character, and there are 96 printable characters (the number on a standard keyboard) then a 7-bit character word length (7 bpc) is required. The entropy, assuming even probabilities (*p*=1/96), is 6.6 bpc.

Table (3.1)
Information content of the English text using different approaches.

#	Approach	bpc
1.	Standard text (7 bpc)	6.6
2.	Entropy	4.5
3.	Huffman code (Average)	4.7
4.	Entropy (Group of 8 characters)	2.4
5.	Asymptotically approaches	1.3

If a probability distribution (based on a corpus of English text) is given for the characters the entropy is reduced to about 4.5 bps. If a separate code is used for each character (for which the Huffman code is optimal), then the number is slightly larger 4.7 bpc

It is clear that so far no advantage, of relationships among adjacent or nearby characters, is considered. If a text is broken into blocks of 8 characters, and the entropy of those blocks (based on measuring their frequency in an English corpus) is measured, then entropy of about 19 bits is obtained. Thus, since 8 characters are coded at a time, the entropy is 2.4 bpc.

If groups of larger and larger blocks are processed, entropy would approach 1.3 (or lower) can be reached. It is impossible to actually measure this because there are too many possible strings to run statistics on, and no corpus is large enough.

This value 1.3 bpc is an estimate of the information content of the English text. Assuming it is approximately correct, this bounds how much can be expected if an English text is losslessly compressed. Table (3.2) shows the compression rate of various data compression algorithms implemented by standard software. All these software, however, are for general purposes and not designed specifically for the English text.

Table (3.2) Information content of the English text using different standard software.		
#	Software	bpc
1.	Compress	3.7
2.	GZIP	2.7
3.	BOA	2.0

The Bayesian Optimization Algorithm (BOA) is the current state-of-the-art for general-purpose compressors. To reach 1.3 bpc, the compressor would surely have to know about English grammar, standard idioms, etc. A complete set of compression ratios for the Calgary corpus for a variety of data compression algorithms is shown in Table (3.3).

Table (3.3) Lossless compression ratios for text compression Calgary corpus.			
#	Scheme	bpc	Researcher
1.	LZ77	3.94	Ziv and Lempel, 1977
2.	LZMW	3.32	Miller and Wegman, 1984
3.	LZH	3.30	Brent, 1987
4.	MTF	3.24	Moffat, 1987
5.	LZB	3.18	Bell, 1987
6.	GZIP	2.71	-
7.	PPMC	2.48	Moffat, 1988
8.	SAKDC	2.47	Williams

9.	PPM	2.34	Cleary, Teahan, and Witten, 1994
10.	BW	2.29	Burrows and Wheeler, 1995
11.	BOA	1.99	Sutton, 1997
12.	RK	1.89	Taylor, 1999

3.2.The Adaptive Character Coding Format

There are different coding formats that can be used in converting a data file into binary digits. They usually have enormous effects on the entropy of the generated binary sequence, and subsequently affect the compression ratio and the coding rate that can be achieved by a particular data compression algorithm over the compressed data file.

A conventional coding format is the ASCII code, in which each character within the source file is coded using an 8-bit codeword. However, a text character is usually coded using 7-bit codeword.

Thus, the length of the binary sequence in bits that is generated from encoding a text file into binary sequence is given by:

$$S_o = 7 \bullet T_c \tag{3.3}$$

Where S_o is the length of the binary sequence in bits, and T_c is the total number of characters within the text file. Another coding format is the Huffman coding, which is described in detail in [Hay 08]. Using Huffman coding, the length of the binary sequence may be expressed as:

$$S_o = T_c \sum_{i=1}^{N_c} f_i \, w_i \tag{3.4}$$

Where N_c is the types of character within the source file.

f_i is the frequency or the probability of occurrence of the i^{th} character.

w_i is the number of bits representing the i^{th} character.

T_c is the total number of characters within the source file (size of file in Bytes).

S_o is the length of the binary sequence generated in bits.

In this thesis, a new coding format is introduced and investigated, namely, the adaptive coding format. In adaptive coding, first, the character frequencies are calculated and sorted in descending order from the most common character to the least, similar to Huffman coding. Second, the most common character is given a 0 sequence number, while the least common character is given N_c-1 sequence number. Then, each character is coded to binary according to its sequence number. For example, the equivalent binary codes for the most (first), second, and the third characters are 0000000, 0000001, and 0000010, respectively.

This form of coding ensures a low entropy binary sequence, therefore, we expect a higher compression ratio and lower bpc is required to represent characters within the compressed file.

In order not to get the data mixed up during the decompression phase, the number of the sorted characters and the characters themselves should be included in the compressed file header. This of course will add an overhead of not more than 129 bytes. It is clear that this overhead is small as compared to the size of the data file.

Table (3.4) presents the binary codeword of the 10 most common characters in the paper1 text file from the Calgary corpus using ASCII and adaptive coding formats.

It is clear that the entropy using adaptive coding will be lower than using ASCII coding as the number of 0s will overwhelm the number of 1s in the binary sequence.

The entropies of the binary sequences generated for a number of text files from the Calgary corpus, using different coding formats (e.g., ASCII coding, Huffman coding, and adaptive coding) are compared in Table (3.5). The results obtained reveal that the adaptive coding has the minim entropy as compared to the others, therefore we expect to achieve higher compression ratio by using this coding format.

Table (3.4)
ASCII and adaptive codes of the 10 most common characters in paper1 file.

#	Character	Frequency	ASCII coding		Adaptive Coding	
			Decimal	Binary	Decimal	Binary
1	Space	7301	32	010000	0	000000
2	e	4689	101	1100101	1	0000001
3	t	3048	116	1110100	2	0000010
4	i	2879	105	1101001	3	0000011
5	o	2568	111	1101111	4	0000100
6	n	2503	110	1101110	5	0000101
7	a	2441	97	1100001	6	0000110
8	s	2374	115	1110011	7	0000111
9	r	2058	114	1110010	8	0001000
10	l	1593	108	1101100	9	0001001

3.3. Concepts of Bit-Level Data Compression Algorithms

In order to use a bit-level data compression algorithm, first, the data file should be represented in binary digits. A data file can be represented in binary digits by concatenating the binary sequences

Table (3.5)
Entropies of the binary sequences generated for a number of text files from the Calgary corpus using different coding formats.

#	File Name	Size (Byte)	N_c	Entropy		
				ASCII	Huffman	Adaptive
1	bib	111261	81	0.999717	0.996537	0.859069
2	book1	768771	82	0.999438	0.995545	0.793412
3	book2	610856	96	0.999078	0.996209	0.818262
4	paper1	53161	95	0.999482	0.996193	0.834678
5	paper2	82199	91	0.998521	0.995549	0.801658
6	paper3	46526	84	0.997612	0.996422	0.810520
7	paper4	13286	80	0.999019	0.995512	0.809544
8	paper5	11954	91	0.999971	0.996128	0.824279
9	paper6	38105	93	1.000000	0.996432	0.835081

of the characters within the file using a specific mapping or coding format, such as ASCII, Huffman and adaptive coding formats. Afterwards, a bit-level processing can be performed to reduce the size of the data files.

The coding format has a huge influence on the entropy of the generated binary sequence and consequently the compression ratio (C) or the coding rate (C_r) that can be achieved.

Usually, in bit-level data compression algorithms, the binary sequence is subdivided into groups of bits that are called minterms, blocks, sub sequences, etc. In this work we shall use the term blocks to refer to each group of bits. These blocks might be considered as representing a Boolean function. Then, algebraic simplifications for bit-reduction are performed on these Boolean functions to reduce the size or the number of blocks, and hence, the number of bits representing the data file is reduced as well.

Examples of bit-level data compression algorithms that are based on this approach include: the Adaptive Character Wordlength (ACW(*n*)) algorithm [Bah 08b], the Adaptive Character Wordlength (ACW(*n*,*s*)) scheme [Bah 08b, Hay 08], the logic-function simplification algorithm [Nof 07], the neural network based algorithm [Mah 00], and the error-correcting Hamming code (HCDC) algorithm [Bah 08a].

3.4. The HCDC Algorithm

The error-correcting Hamming code has been widely used in computer networks and digital data communication systems as a single bit error correcting code or two bits errors detection code. It can also be tricked to correct burst errors. The key to Hamming code is the use of extra parity bits (*p*) to allow the identification of a single bit and a detection of two bits errors [Kim 05, Tan 03].

Thus, for a message having *d* data bits and to be coded using Hamming code, the coded message (also called codeword) will then have a length of *n* bits, which is given by:

$$n = d + p \tag{3.5}$$

This would be called a (*n*,*d*) code. The optimum length of the codeword (*n*) depends on *p*, and it can be calculated as:

$$n = 2^{p} - 1 \tag{3.6}$$

The data and the parity bits are located at particular locations in the codeword. The parity bits are located at positions 2^0, 2^1, 2^2, …, 2^{p-1} in the coded message, which has at most *n* positions. The remaining positions are reserved for the data bits, as shown in Figure (3.1). Each parity bit is computed on different subsets of the data bits, so that it forces the parity of some collection of data bits, including itself, to be even or odd.

A lossless binary data compression algorithm based on the error correcting Hamming codes, namely the HCDC algorithm, was proposed by H. Al-Bahadili [Bah 08a]. In this algorithm, the data symbols (characters) of a source file are converted to binary sequence by concatenating the individual binary codes of the data symbols.

The binary sequence is, then, subdivided into a number of blocks, each of *n*-bit length as shown in Figure (3.1b). The last block is padded with 0s if its length is less than *n*. For a binary sequence of S_o bits length, the number of blocks *B* (where *B* is a positive integer number) is given by:

$$B = \left\lceil \frac{S_o}{n} \right\rceil \qquad (3.7)$$

The number of padding bits (*g*), which may be added to the last block is calculated by:

$$g = B * n - S_o \qquad (3.8)$$

The number of parity bits (*p*) within each block is given by:

$$p = \left\lceil \frac{\ln(n+1)}{\ln(2)} \right\rceil \qquad (3.9)$$

For a block of n-bit length, there are 2^n possible binary combinations (codeword) having decimal values ranging from 0 to 2^n-1, only 2^d of them are valid codewords and 2^n-2^d are non-valid codewords.

Each block is then tested to find if it is a valid block (valid codeword) or a non-valid block (non-valid codeword). During the compression process, for each valid block the parity bits are omitted, in other words, the data bits are extracted and written into a temporary compressed file. However, these parity bits can be easily retrieved back during the decompression process using Hamming codes. The non-valid blocks are stored in the temporary compressed file without change.

In order to be able to distinguish between the valid and the non-valid blocks during the decompression process, each valid block is preceded by 0, and each non-valid block is preceded by 1 as shown in Figure (3.1c). Figures (3.2) and (3.3) summarize the flow of the compressor and the decompressor of the HCDC algorithm.

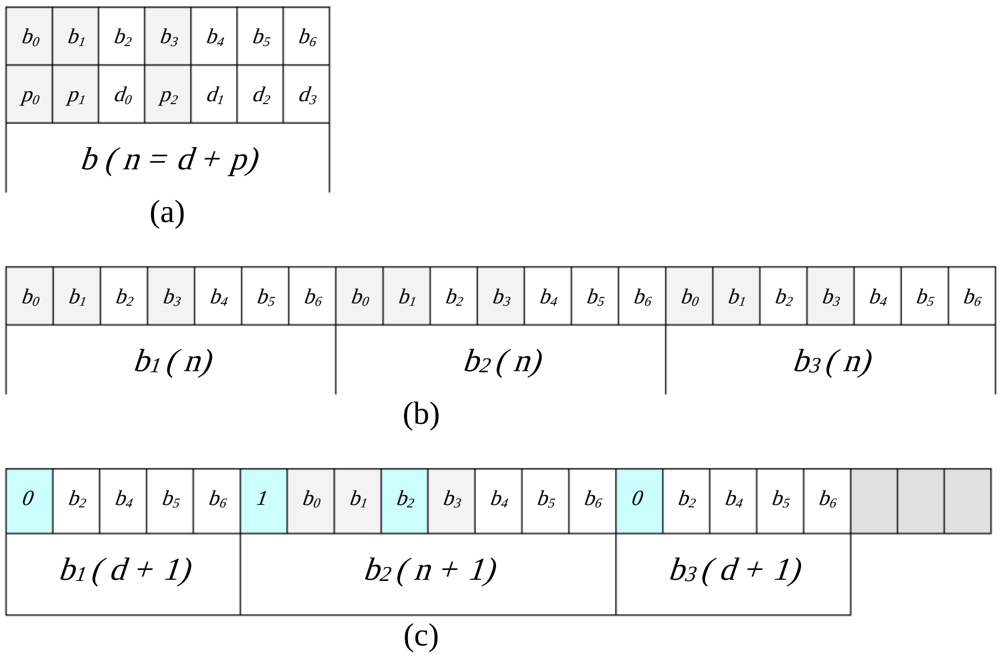


Figure (3.1) - (a) Locations of data and parity bits in 7-bit codeword, (b) an uncompressed binary sequence of 21-bit length divided into 3 blocks of 7-bit length, where b_1 and b_3 are valid blocks, and b_2 is a non-valid block, and (c) the compressed binary sequence (18-bit length).

```
1.   Initialization
        Select $p$
        Calculate $n = 2^p - 1$
        Calculate $d = n - p$
        Calculate $B = \text{ceiling}(S_o/n)$
        Calculate $g = B * n - S_o$
        Initialize $b = 0$
2.   Reading binary data
        Read a block of $n$-bit length
        [Add 1 to $b$]
3.   Check for block validity
        If  (block = valid codeword) then
            Add 1 to □
            Extract the data bits ($d$-bit)
            Write 0 followed by the extracted $d$-bits to the
            temporary compressed file
        Else (block = non-valid codeword)
            Add 1 to □
            Write 1 followed by all $n$-bits to the temporary
            compressed file
        End if
4.   If ($b<B$) then Goto Step 2
```

Figure (3.2) - The main steps of the HCDC compressor.

```
1.  Initialization
        Select $p$
        Calculate $n = 2^p - 1$
        Calculate $d = n - p$
        Initialize $b = 0$
2.  Reading binary data
        Read one bit ($h$)
        [Add 1 to $b$]
3.  Check for block validity
        If  {$h = 0$} then
          [Add 1 to $v$□
          Read the following $d$ data bits
          Compute the Hamming codes for these $d$ data bits
          Write the coded block the temporary decompressed binary
          sequence
        Else {$h = 1$} then
          [Add 1 to $w$]
          Read a block of $n$ bits length
          Write $n$ bits block to the  temporary decompressed  binary
          sequence
        End if
4.  If (not end of data) go to Step 2
```

Figure (3.3) - The main steps of the HCDC decompressor.

3.4.1 Derivation and Analysis of HCDC Algorithm Compression Ratio

This section presents the analytical derivation of a formula that can be used to compute the compression ratio achievable using the HCDC algorithm. The derived formula can be used to compute *C* as a function of two parameters:

i. The block size (*n*).

ii. The fraction of valid blocks (*r*).

In the HCDC algorithm, the original binary sequence is divided into *B* blocks of *n*-bit length. These *B* blocks are either valid or non-valid blocks; therefore, the total number of blocks is given by:

$$B = v + w \tag{3.10}$$

where v and w are the number of valid and non-valid blocks, respectively.

As it has been discussed in the previous section, in the HCDC algorithm, the binary sequence is divided into a number of blocks of n-bit length, each block is then subdivided into d data bits and p parity bits. For a valid block only the d data bits preceded by 0 are appended to the compressed binary sequence (i.e., d+1 bits for each valid block). So that the length of the compressed valid blocks (S_v) is given by:

$$S_v = v\,(d + 1) \tag{3.11}$$

For a non-valid block all bits are appended to compressed binary sequence (i.e., n+1 bits for each non-valid block). The number of bits appended to the compressed binary sequence is given by:

$$S_w = w\,(n + 1) \tag{3.12}$$

Thus, the length of the compressed binary sequence (S_c) can be calculated by:

$$S_c = S_v + S_w = v\,(d + 1) + w\,(n + 1) \qquad (3.13)$$

Using Eqns. (3.5) and (3.10), Eqn. (3.13) can be simplified to

$$S_c = Bn + B - v\square p$$

(3.14)

Substituting S_o=nB and S_c as it is given by Eqn. (3.14) into the equation of the compression ratio (C) yields:

$$C = \frac{S_o}{S_c} = \frac{n}{n+1-r\;p} \qquad (3.15)$$

where r=v/B, and it represents the fraction of valid blocks. Substitute Eqn. (3.6) into Eqn. (3.15) gives:

$$C = \frac{2^p - 1}{2^p - r\;p} \qquad (3.16)$$

It is clear from Eqn. (3.16) that, for a certain value of p, C is inversely proportional to r, and C is varied between a maximum value (C_{max}) when r=1 and a minimum value (C_{min}) when r=0. It can also be seen from Eqn. (3.16) that for each value of p, there is a value of r at which C=1. This value of r is referred to as r_1, and it can be found that r_1=1/p.

Table (3.6) lists the values of C_{max}, C_{min}, and r_1 for various values of p. These results are also shown in Figures (3.4) and (3.5), where Figure (3.4) shows the variation of C_{max} and C_{min} with p, and Figure (3.5) shows the variation of r_1 with p. Figure (3.6) shows the variations of C with respect to r for values of p varies between 2 to 8.

The numerical results are tabulated in Table (3.7). It can be deduced from Figure (3.6) and Table (3.7) that satisfactory values of C can be achieved when $p \leq 4$ and $r > r_1$.

Finally, one important feature of the HCDC algorithm is that it can be repeatedly applied on the binary sequence, and an equation can be derived to compute, what we refer to as the accumulated compression ratio (C_k):

$$C_k = \prod_{i=1}^{k} C_i = \prod_{i=1}^{k} \frac{S_{i-1}}{S_i} \qquad (3.17)$$

Where k is the number of repetitions; S_i and S_{i-1} are the sizes of the binary file before and after the i^{th} compression loop; C_i is the compression ratio of the i^{th} compression loop. For i=1, S_o represents the size of the original file.

Table (3.6) Variation of C_{min}, C_{max}, and r_1 with number of parity bits (p).			
p	C_{min}	C_{max}	r_1
2	0.750	1.500	0.500
3	0.875	1.400	0.333
4	0.938	1.250	0.250
5	0.969	1.148	0.200
6	0.984	1.086	0.167
7	0.992	1.050	0.143
8	0.996	1.028	0.125

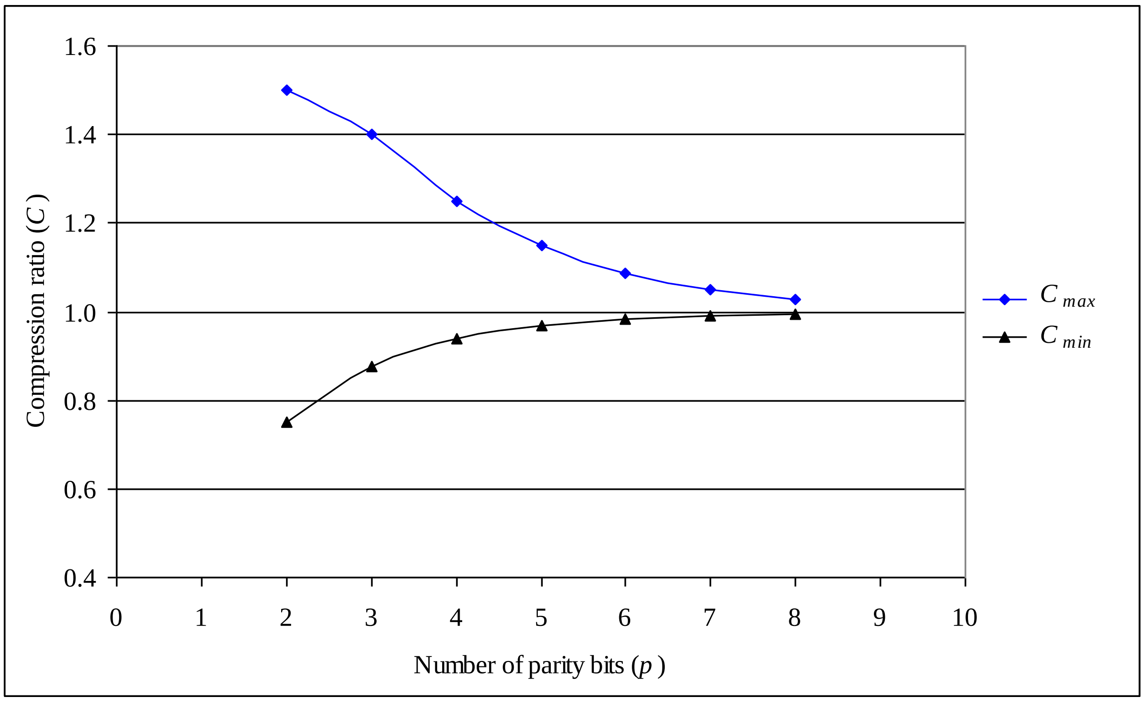


Figure (3.4) - Variation of C_{min} and C_{max} with p.

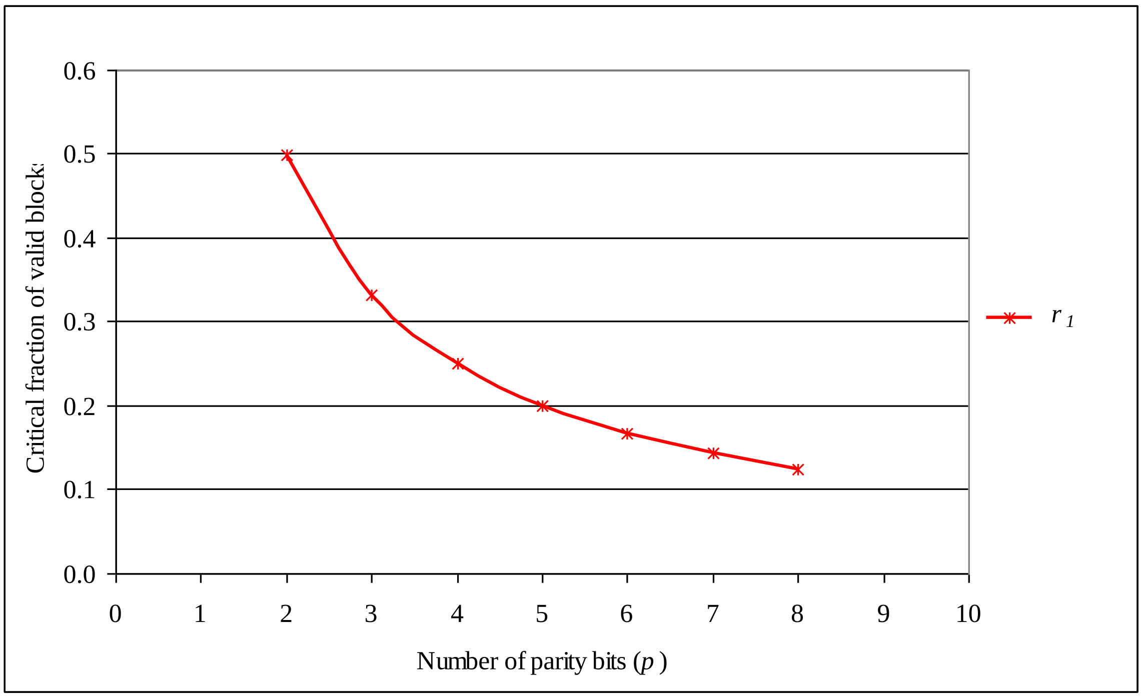


Figure (3.5) - Variation of r_1 with p.

Table (3.7) Variations of C with respect to r for various values of p.							
r	Number of the parity bits (p)						
	2	3	4	5	6	7	8
0.0	0.750	0.875	0.938	0.969	0.984	0.992	0.996
0.1	0.789	0.909	0.962	0.984	0.994	0.998	0.999
0.2	0.833	0.946	0.987	1.000	1.003	1.003	1.002
0.3	0.882	0.986	1.014	1.016	1.013	1.009	1.006
0.4	0.938	1.029	1.042	1.033	1.023	1.014	1.009
0.5	1.000	1.077	1.071	1.051	1.033	1.020	1.012
0.6	1.071	1.129	1.103	1.069	1.043	1.026	1.015
0.7	1.154	1.186	1.136	1.088	1.054	1.032	1.018
0.8	1.250	1.250	1.172	1.107	1.064	1.038	1.022
0.9	1.364	1.321	1.210	1.127	1.075	1.044	1.025

1.0	1.500	1.40	1.250	1.148	1.086	1.050	1.028

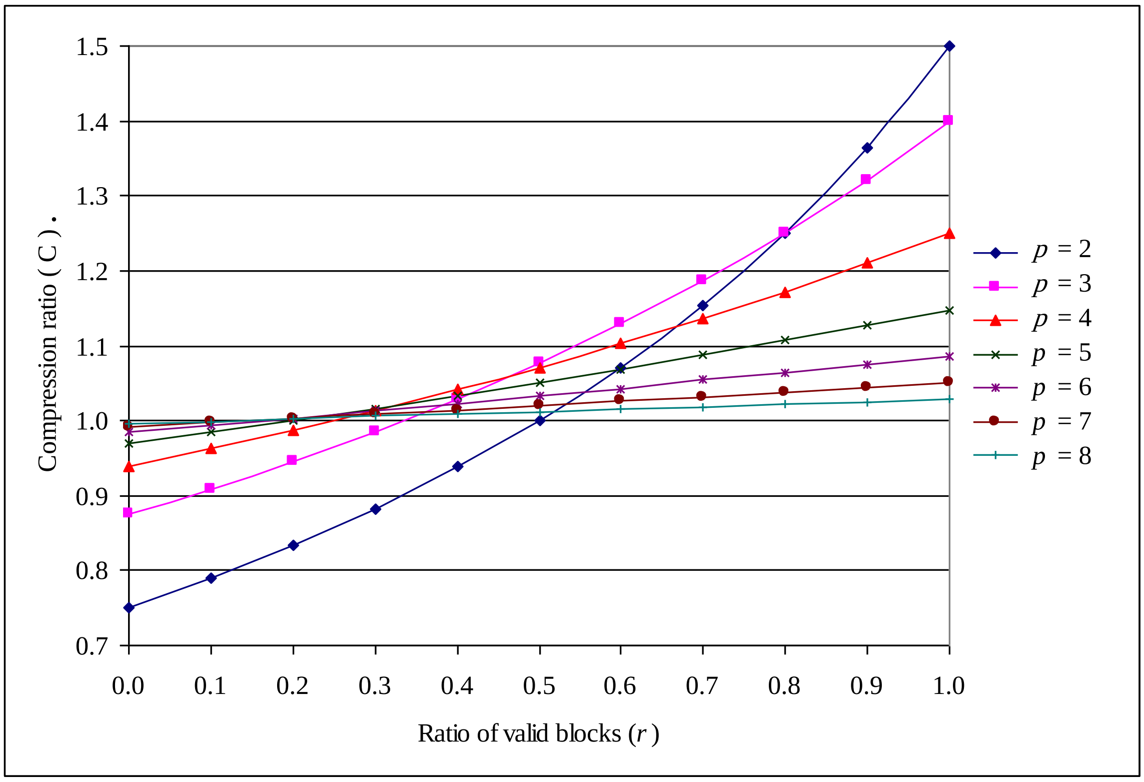


Figure (3.6) - Variations of *C* with respect to *r* for various values of *p*.

English text characters are usually converted to binary using its equivalent 7-bit ASCII codes, which means that each character can be considered as a (7,4) codeword. It has been mentioned earlier that not all 7-bit codewords are valid codewords, in fact only 16 codewords (2^d) are valid, and the remaining 112 codewords (2^n-2^d) are non-valid. The ASCII codes of these 16 valid codewords and the characters they represent are listed in Table (3.8).

According to the statistics in Standard English text, these valid codewords can be categorized into three groups:

i. Wide-use (their decimal values are equivalent to the ASCII code of the characters a and f).

ii. Rare-use (their decimal values are equivalent to the ASCII code of the characters K, L, R, U, x, *, -, 3, and 4).

iii. Not-used (their decimal values are equivalent to the ASCII code of the unprintable characters of 0, 7, 25, 30, and 127 ASCII code).

Table (3.8) – Valid 7-bit codewords.

ASCII	Character	Category
0	Control Character	Not used
7	Control Character	Not used
25	Control Character	Not used
30	Control Character	Not used
42	*	Rare-use
45	-	Rare-use
51	3	Rare-use
52	4	Rare-use
75	K	Rare-use
76	L	Rare-use
82	R	Rare-use
85	U	Rare-use

97	a	Frequent -
102	f	Frequent-
120	x	Rare-use
127	Control Character	Not-used

Thus, encoding characters to binary using their equivalent ASCII codes and testing the validity of these characters either yields a very low compression ratio or most properly yields inflation. This is because a small proportion of the characters within the text may have valid codewords.

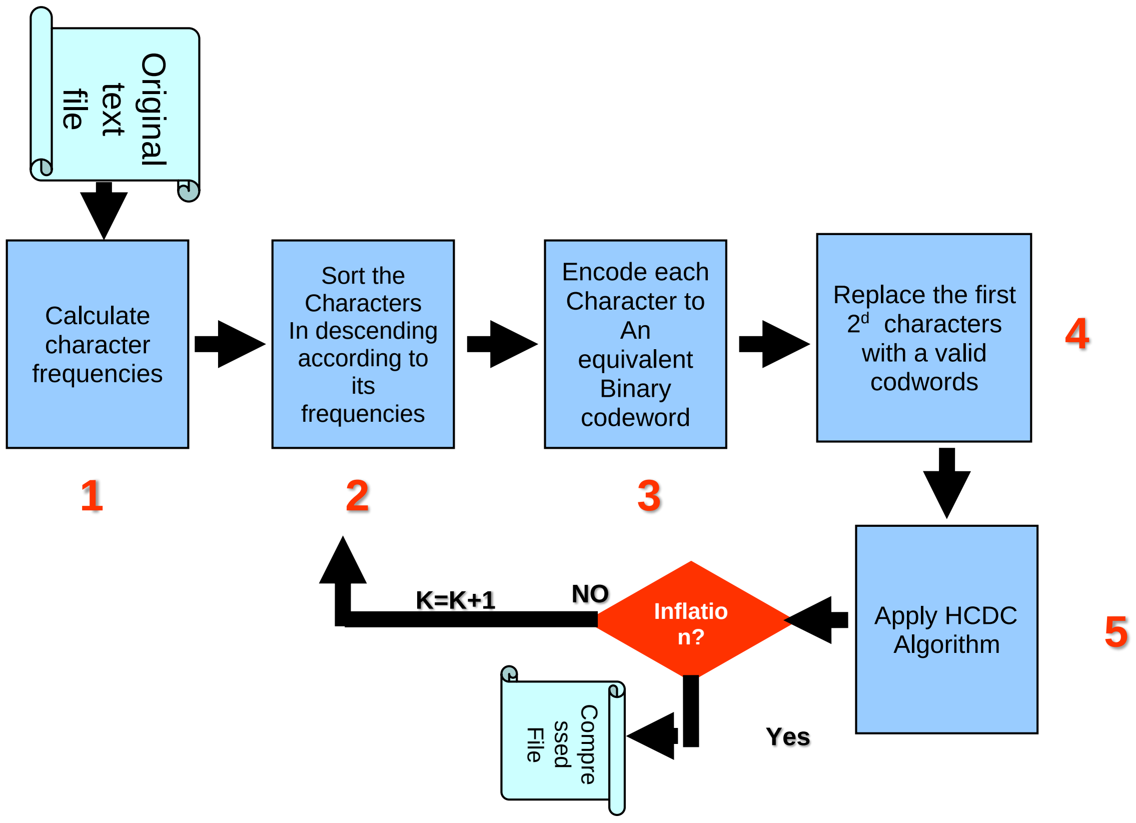
If adaptive coding is used in converting the text characters into binary sequence, it produces low entropy binary sequence, but still only two of these sequence codes are valid codewords (0 and 7), which represent the 1st and the 8th most common characters. Once again, these two codes may represent a small proportion of the characters within the text file, so that achieving reduction in the size of the text file is critical.

3.5. The HCDC (k) Scheme

As it has been discussed in the previous section, a straight forward implementation of the HCDC algorithm for text compression may not always produce an adequate compression ratio regardless of the coding format that is used to convert the text characters to binary sequence

Therefore, to enhance the performance of the HCDC algorithm for text compression, we develop an enhanced scheme. The new scheme is based on the HCDC algorithm and it consists of six main steps, which are repetitively applied to achieve higher compression ratio [Bah 07a].

The repetition loops continue until inflation detected and the overall compression ratio is the multiplication of the compression ratios of the individual loops, therefore we refer to the new scheme as HCDC(*k*), where *k* refers to the number of repetition loops. Figure (3.7) represent the flowchart of the HCDC(*k*).



Original text file
Calculate character frequencies
1
Sort the Characters In descending according to its frequencies
2
Encode each Character to An equivalent Binary codeword
3
Replace the first 2d characters with a valid codwords
4
Apply HCDC Algorithm
5
Inflation?
NO
K=K+1
Yes
Compressed File

Figure (3.7) – The HCDC(*k*) Scheme flowchart.

In order to enhance the compression power of the HCDC(*k*) scheme, the adaptive coding format is used in which a text character is encoded to binary according to its frequency so that the binary sequence will have low entropy and higher compression ratio is granted.

The six main steps of the HCDC(*k*) scheme can be summarized as follows:

Step 1: Calculate characters frequencies (f_i, i=1, 2, 3, …, N_c, where N_c is the number of symbols within the text file). At this stage, if the standard HCDC algorithm is performed, the resulting compression ratio for a single loop can be derived as:

$$C = n\sum_{i=1}^{N_c} f_i \Bigg/ \left((d+1) \sum_{\substack{i=1 \\ A_i = V_c}}^{N_c} f_i + (n+1) \sum_{\substack{i=1 \\ A_i \neq V_c}}^{N_c} f_i \right) \qquad (3.18)$$

where A_i is the ASCII code of the i^{th} character, and V_c is any valid 7-bit codeword. In the above equation, $\sum_{i=1}^{N_c} f_i$ represents the total number of characters (number of blocks (*B*) within the text file). While $\sum_{\substack{i=1 \\ A_i = V_c}}^{N_c} f_i$ and $\sum_{\substack{i=1 \\ A_i \neq V_c}}^{N_c} f_i$ represent the number of valid characters (*v*) and non-valid characters (*w*), respectively.

For a (7, 4) Hamming code, it can be easily shown that for a compression to be achieved (*C*>1) the fraction of valid characters (*r*=*v*/*B*) should be greater than 1/3. The compression ratio is directly proportional to *r*, and the maximum compression ratio that can be achieved is 1.4 (7/5) when all characters have valid codewords.

Step 2: Sort characters in descending order according to their frequencies. Start with 0 for the most common character and N_c-1 for the least common character. A list of these sorted characters is stored in the compressed file header, to be used during the decompression process.

Step 3: Encode each character within the text file to an equivalent binary digits. Each character in the text file is converted to 7-bit binary digits according to its sequence number, not according to its equivalent ASCII code. For Standard English text files, this of course will reduce the entropy of the binary file and grants higher compression ratio.

Step 4: Replace any of the 16 most common codewords with a valid codeword if it is not originally representing a valid codeword.

A record of each original codeword and its replacement must also be recorded and stored in the compressed file header. In this case, the compression ratio can be expressed as:

$$C = n\sum_{i=1}^{N_c} f_i \Bigg/ \left((d+1)\sum_{i=1}^{2^d} f_i + (n+1)\sum_{i=17}^{N_c} f_i \right) \qquad (3.19)$$

Step 5: Test each 7-bit block to find if it is a valid or a non-valid codeword. If the codeword is valid, then only the data bits (bits at positions 3, 5, 6, and 7) preceded by 0 are written to the compressed binary file, otherwise all bits preceded by 1 are written to the file.

Step 6: Repeat steps 2 to 5 until inflation (C<1) is detected. The compressed binary file of each loop can be repeatedly processed until the compression ratio either reaches steady state (i.e., saturated) or inflation is detected. This may occur because after a few iterations the 7-bit binary blocks will have equal frequencies.

The overall or the accumulated compression ratio (C_k) is computed as a multiplication of the compression ratio of the individual loops (C_i), as expressed in Eqn. (3.17).

The main features of the HCDC(*k*) scheme are:

i. Bit-level. It processes the text file at a binary level.

ii. Lossless. An exact form of the source file can be retrieved.

iii. Adaptive. The character-binary coding depends on the characters frequencies.

iv. Symmetric. The compression/decompression CPU times are almost the same.

3.6 The Compressed File Header

The HCDC(*k*) scheme compressed file header contains all additional information that is required by the decompression algorithm. It consists of the following fields:

i. HCDC field

ii. Coding filed

iii. Valid codewords field

iv. Replacement field

v. Padding field

Following is a brief description for each of the above fields

i. HCDC field

HCDC field is an 8-byte field that encloses information related to the HCDC(*k*) scheme, such as: algorithm name (HCDC), algorithm version (*V*), coding format (*F*), number of compression loops (*k*).

The original HCDC algorithm was designated as Version-0 (i.e., *V* is set to 0), the HCDC(*k*) scheme is designated as Version-1 (i.e., *V* is set to 1).

The coding format (*F*) is set to 0 for ASCII coding, 1 for Huffman coding, 2 for adaptive coding, etc. Table (3.9) lists the constituents of this field and their description.

ii. Codes field

The codes field encloses information related to the coding format, so that their content depends on the coding format indicated in the HCDC field. This field is not required for the ASCII coding format. For the adaptive coding, it contains the symbols within the source file sorted descendingly according to their frequency of occurrence. The length of this field is N_c bytes. For Huffman coding, it enfolds the same components as in standard Huffman compression algorithm.

iii. Valid codewords field

This field encloses the 16 valid codewords to be assigned for the most frequently used characters. The length of this field is constant 16 Byte.

iv. Replacement field

The replacement field contains information on the original codewords and their replacements. The length of this field is 2R_i bytes for each loop. The maximum value of R_i is 16, if all most common 16 codewords are non-valid and need to be replaced with valid codewords.

iv. Padding field

This field contains information on the number of padding bits (*g*) during each compression loop from loop 1 to loop *k*. Thus, the length of this field is *k* Bytes.

Taking into account all fields mentioned above, the length of the HCDC(*k*) scheme compressed file header (H_l) can be expressed as:

$$H_l = 8 + N_c + 16 + 2\sum_{i=1}^{k} R_i + k \qquad (3.20)$$

The fields of the compressed file header and their individual lengths in Bytes are shown in Figure (3.9).

Table (3.9)
HCDC(k) scheme compressed file header.

Field	Length (Byte)	Description
HCDC	4	Name of the compression algorithm.
V	1	Version of the HCDC algorithm.
N_c	1	Number of symbols within the original text file.
F	1	Coding format (0 for ASCII coding, 1 for Huffman coding, 2 for adaptive coding, etc.)
k	1	The number of compression loops.

HCDC field (8 Byte)	Codes field (N_c Byte)	Valid codewords field (16 Byte)	Replacements field ($2\sum_{i=1}^{k} R_i$ Byte)	Padding field (k Byte)

Figure (3.8) - The compressed file header of the HCDC(k) scheme.

Chapter Four
Experimental Results and Discussions

The HCDC(*k*) scheme is implemented using C++ programming language. The resultant code allows a wide range of investigations and experiments to be performed. However, it is only used to compress a number of text files from standard corpora, namely, Calgary corpus, Canterbury corpus, Artificial corpus, and Large corpus, which are described in Appendix A.

At this stage, it is important to indicate that little efforts have been taken to optimize the runtime of the compression/decompression prototype code, therefore, in this work, we only compare and show the results for the compression ratio of the scheme. However, this scheme is classified as an asymmetric bit-level data compression algorithm as the compression processing time is higher than the time required for decompression.

In this thesis, results of five experiments are presented to evaluate and compare the performance of the HCDC(*k*) scheme. In all experiments, the HCDC(*k*) header taken into consideration, these five experiments are described in the next sections.

4.1 Experiment #1: Investigating the Effect of the Character Coding Format

In this experiment we investigate the effect of the character coding formats, namely, the ASCII and the adaptive coding formats on the compression ratios achieved by the HCDC(*k*) scheme. They are investigated using two text files of different sizes from the Calgary corpus, namely, paper1 and book1. While paper1 file is characterized by its small size (53161 Byte), book1 file is characterized by its large size (768771 Byte).

The results obtained for the entropy of the compressed binary file (*E*), loop compression ratio (C_i) and the accumulated compression ratio (C_k) for the text files paper1 and book1, are presented in Tables (4.1) and (4.2), respectively.

The results obtained demonstrate that adaptive coding grants higher compression ratios as compared to ASCII coding for all values of *k* except for *k*=1, where they provide the same compression ratio, because for both coding formats the 16 most common characters were replaced with valid codewords. For *k*=1, the compression ratios achieved by the HCDC(1) scheme (HCDC algorithm) over paper1 and book1 files are 1.211 and 1.269, respectively.

The tabulated results show that initially the entropy of the binary files produced using adaptive coding are less than ASCII coding for both text files; therefore, in general higher C_k are achieved. It can also be seen that for adaptive coding the entropy of the compressed binary file is increasing as the compression process is going on, while it is decreasing for ASCII coding. However, using ASCII coding, the rate of reduction in entropy is very small, thus the advantage it produces is not enough to overcome the inflation in the compression ratio that occurs during the particular compression stage (iteration). Consequently, the accumulated compression ratio decreases after few iterations (k=2).

For adaptive coding, although the entropy is increasing as the compression process going on, the loop compression ratio remains uninflated and the accumulated compression ratio increases steadily. However, inflation begins after a number of iterations (4 to 6).

For paper1, the maximum compression ratio that can be achieved by HCDC(k) scheme using ASCII coding is 1.234 at k=2, while adaptive coding achieves a maximum compression ratio of 1.658 at k=4.

For book1, the HCDC(k) scheme achieves a maximum compression ratio of 1.309 at *k*=2 for ASCII coding, and 2.225 at *k*=4 using adaptive coding. Furthermore, using ASCII coding, the iteration process enhances the compression ratio achieved over paper1 and book1 text files are 2% and 3%, respectively; while using adaptive coding enhances the compression ratio by 40% and 75%. The results for the accumulated compression ratio (C_k) in Tables (4.1) and (4.2) are plotted in Figures (4.1) and (4.2), respectively.

Table (4.1) – Experiment #1

The HCDC(*k*) scheme compression ratio achieved for paper1 file using ASCII and adaptive character coding formats.

k	paper1					
	ASCII Coding			Adaptive Coding		
	E	*C*	C_k	*E*	*C*	C_k
1	0.999	1.211	1.211	0.835	1.211	1.211
2	0.999	1.019	1.234	0.757	1.180	1.429
3	0.998	0.943	1.163	0.846	1.109	1.585
4	0.990	0.946	1.101	0.929	1.046	1.658
5	0.984	0.947	1.042	0.984	1.000	1.657
6	0.974	0.954	0.995	1.000	0.977	1.619
7	0.971	0.957	0.952	0.998	0.958	1.551

8	0.963	0.966	0.920	0.991	0.949	1.472
Shaded cells represent highest compression ratio						

Table (4.2) – Experiment #1
The HCDC(*k*) scheme compression ratio achieved for book1 file using ASCII and adaptive character coding formats.

k	book1					
	ASCII Coding			Adaptive Coding		
	E	*C*	C_k	*E*	*C*	C_k
1	0.999	1.269	1.269	0.793	1.269	1.269
2	0.999	1.031	1.309	0.649	1.246	1.582
3	0.998	0.948	1.240	0.727	1.184	1.873
4	0.992	0.940	1.166	0.824	1.119	2.096
5	0.983	0.945	1.102	0.914	1.056	2.213
6	0.974	0.955	1.053	0.976	1.005	2.225
7	0.965	0.964	1.015	0.999	0.972	2.163
8	0.962	0.969	0.983	0.998	0.952	2.060

Shaded cells represent highest compression ratio

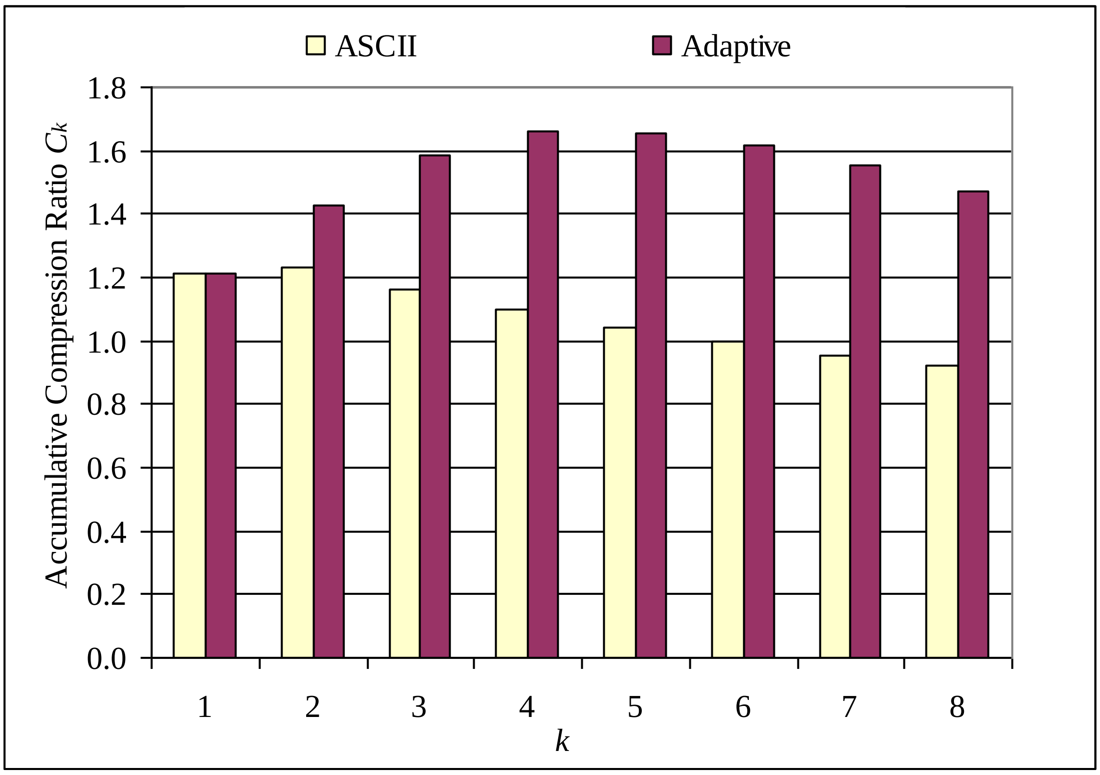


Figure (4.1) - Comparison of C_k for ASCII and adaptive coding formats for paper1 file.

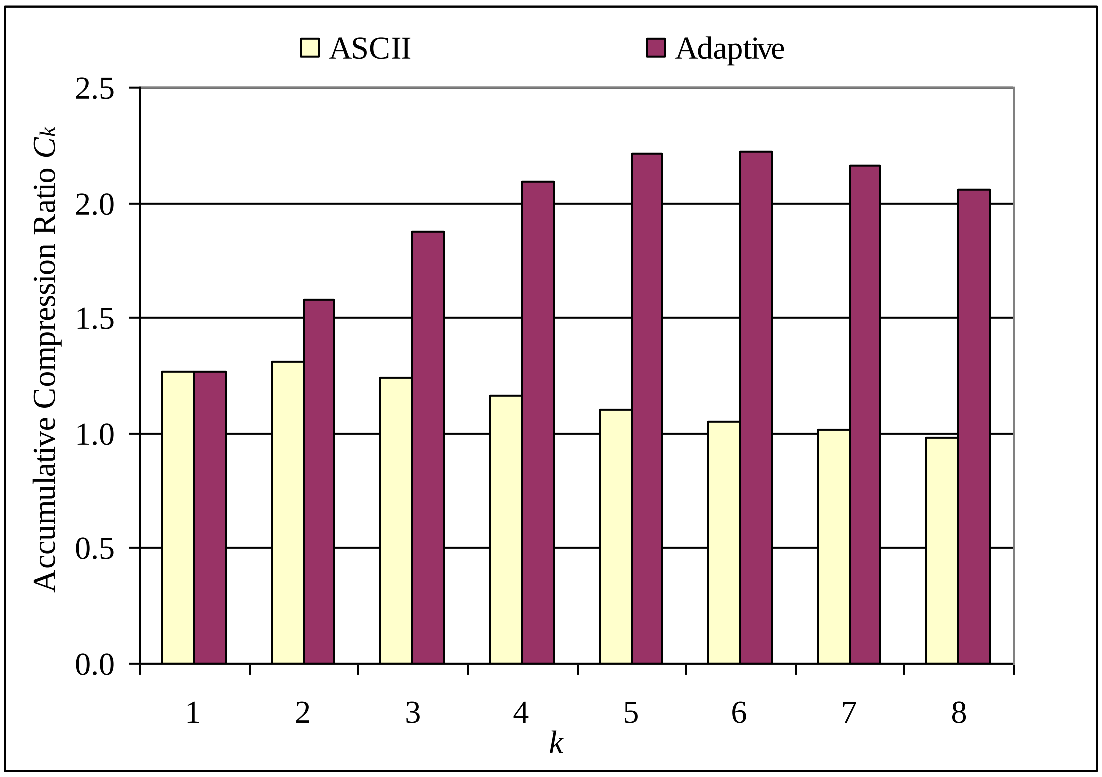

□ ASCII
■ Adaptive
Accumulative Compression Ratio C_k
2.5
2.0
1.5
1.0
0.5
0.0
1
2
3
4
5
6
7
8
k

Figure (4.2) - Comparison of C_k for ASCII and adaptive coding formats for book1 file.

4.2. Experiment #2: Evaluating the Compression Ratio of the HCDC (k) Scheme over a Number of Text Files from Standard Corpora

In the following experiments, the coding format that will be used to convert a text file into a binary file is the adaptive coding, because it can provide higher compression ratio than ASCII coding. Experiment #2 evaluates and compares the compression ratio of the HCDC(k) scheme over a number of text files from standard corpora, namely, Calgary Corpus, Canterbury Corpus, Artificial Corpus, and Large Corpus. The results obtained are tabulated in Table (4.3). The results include values of the following parameters:

i. Name and size of the text file.

ii. Number of characters within the text file (N_c).

iii. The compression ratio and the length of the compressed file header for the HCDC(1) scheme, C_1 and H_1, respectively.

iv. The compression ratio and the length of the compressed file header for the HCDC(k) scheme, C_k and H_k, respectively.

v. The enhancement ratio (E_k), which is calculated by:

$$E_k = \frac{C_k - C_1}{C_1} \times 100 \qquad (4.1)$$

where C_k and C_1 are the compression ratios of the HCDC(*k*) scheme after *k* loops and after the first loop (*k*=1), respectively.

The compression ratios of the HCDC(*k*) scheme achieved for text files from the Calgary corpus are shown in Figure (4.3), while Figure (4.4) shows the compression ratios achieved for text files from the Canterbury, Artificial, and Large corpora.

It is also important to realize that the compression ratio of the HCDC(1) scheme can be calculated analytically using Eqn. (3.19) after calculating the character frequencies, and this can be used to validate the accuracy of the coding process.

Table (4.3) – Experiment #2

Comparison between C_1 and C_k for a number of text files from the Calgary corpus.

#	File Name	File Size (Byte)	N_c	HCDC(1) scheme		HCDC(*k*) scheme		E_r %
				C_1	H_1	C_k	H_k	

Calgary corpus								
1	Bib	111261	81	1.177	141	1.428 (4)	225	21
2	book1	768771	82	1.269	144	2.225 (6)	290	75
3	book2	610856	96	1.247	157	1.971 (5)	272	58
4	paper1	53161	95	1.211	155	1.658 (4)	239	37
5	paper2	82199	91	1.265	153	2.178(6)	299	72
6	paper3	46526	84	1.261	145	2.112 (5)	260	67
7	paper4	13286	80	1.257	142	2.118 (6)	288	68
8	paper5	11954	91	1.218	152	1.737 (5)	267	43
9	paper6	38105	93	1.203	153	1.597 (4)	237	33
Canterbury corpus								
10	alice29.txt	152089	74	1.258	135	2.097 (5)	250	67
11	asyoulik.txt	125179	68	1.230	129	1.825 (5)	244	48

12	lcet10.txt	426754	84	1.252	145	2.009 (5)	260	60
13	plrabn12.txt	481861	81	1.269	143	2.235 (6)	289	76
Artificial corpus								
14	aaa.txt	100000	1	1.400	33	7.5 (6)	43	436
15	alphabet.txt	100000	26	1.138	86	1.425 (4)	170	25
16	random.txt	100000	64	0.969	121	0.969 (1)	112	0
Large corpus								
16	bible.txt	4047390	63	1.294	125	2.656 (6)	271	105
17	world192.txt	2473400	94	1.206	154	1.621 (4)	238	34

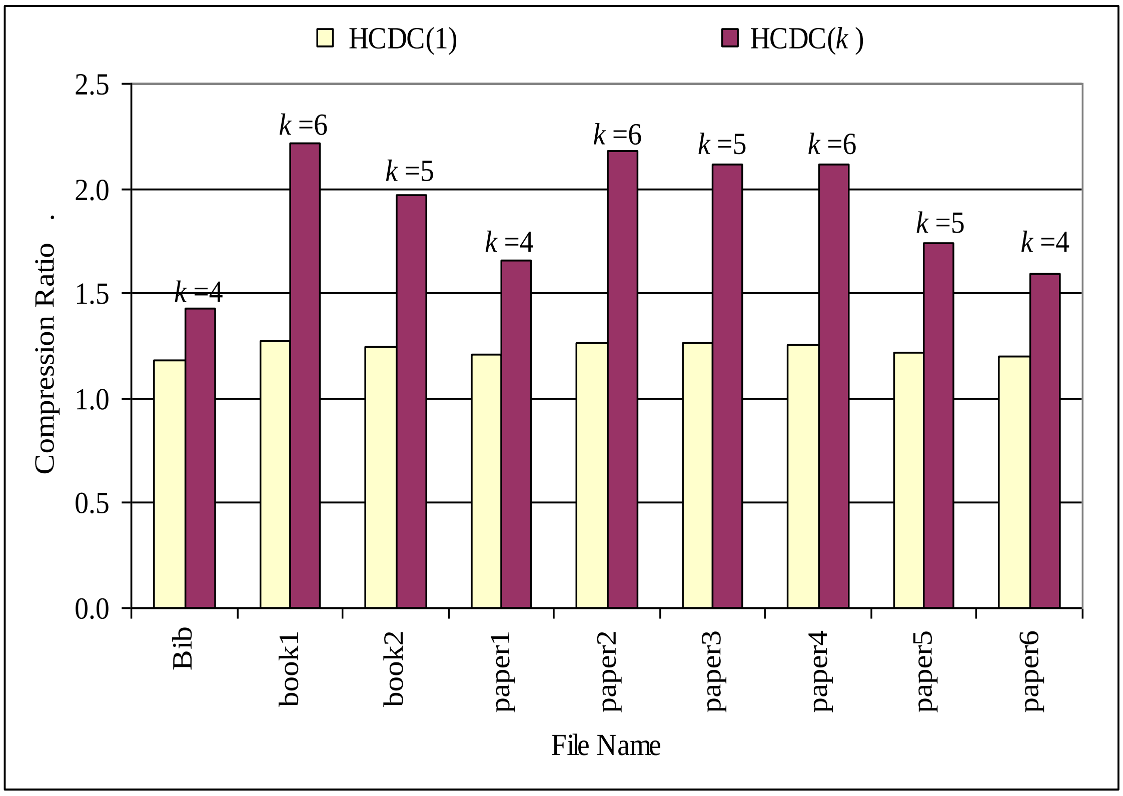


Figure (4.3) – Comparison between C_1 and C_k for a number of text files from the Calgary corpus.

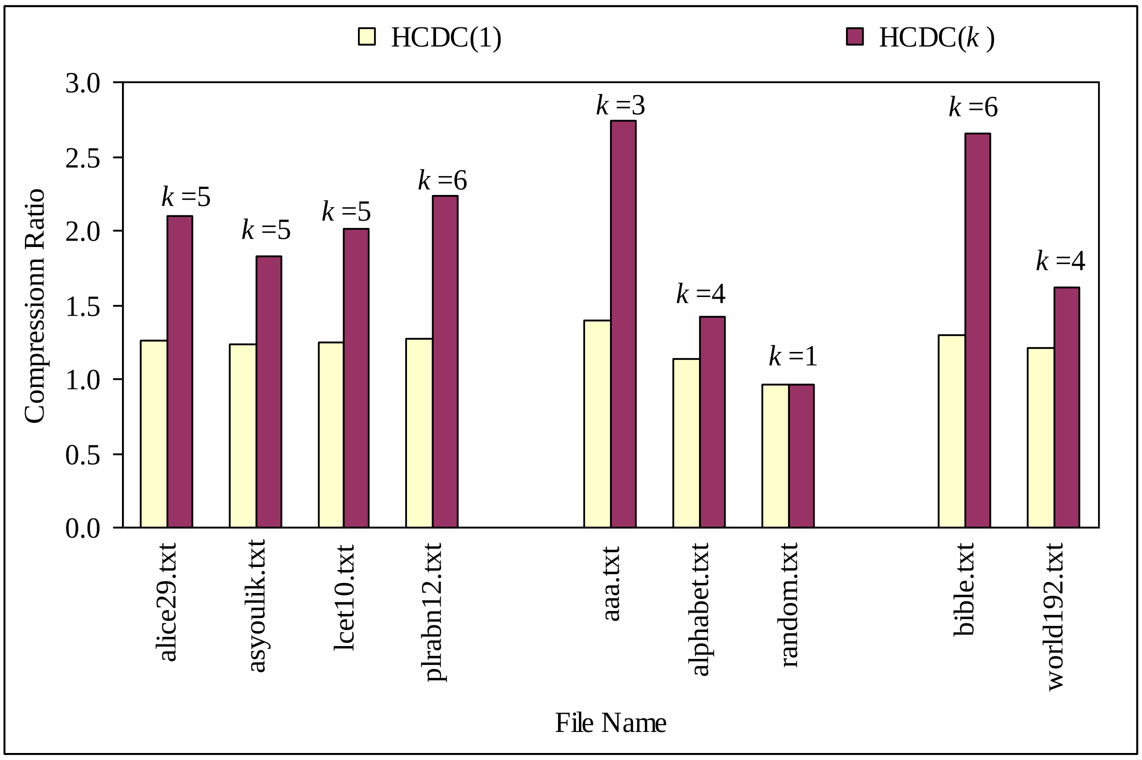


Figure (4.4) - Comparison between C_1 and C_k for a number of text files from the Canterbury, Artificial, and Large corpora .

4.3.Experiment # 3: Comparing the Compression Ratio of the HCDC(k) Scheme with a Number of Data Compression Algorithms

In Table (4.4), the compression power of the HCDC(*k*) scheme is compared with a number of statistical and bit-level data compression algorithms, such as: the Huffman coding (HU), the fixed-length Hamming (FLH), and the Huffman coding following the fixed-length Hamming (HF) [Sha 04], the ACW(*n*,*s*) scheme using adaptive coding (ACW-A), and the ACW(*n*,*s*) scheme using Huffman coding (ACW-H) [Hay 08]. The results show that the HCDC(*k*) algorithm achieves high compression ratios with respect to other algorithms. The results are also presented in Figure (4.5).

Table (4.4) – Experiment #3 Comparison between the compression ratio of the HCDC(*k*) scheme and various statistical and bit-level data compression algorithms.		
Algorithm	Book1	Paper1
HU*	1.72	1.60
FLH*	1.14	1.14
HF*	1.71	1.57
ACW–A**	1.674 (14)	1.542 (11)
ACW–H**	2.673 (11)	2.431 (11)
HCDC(1)	1.27 (1)	1.21 (1)
HCDC(*k*)	2.225 (6)	1.658 (4)

* HU: Huffman coding, FLH: Fixed-Length Hamming, HF: HU following FLH [Sha 04].

** ACW(*n*,*s*): Adaptive Character Word length algorithm [Hay 08].

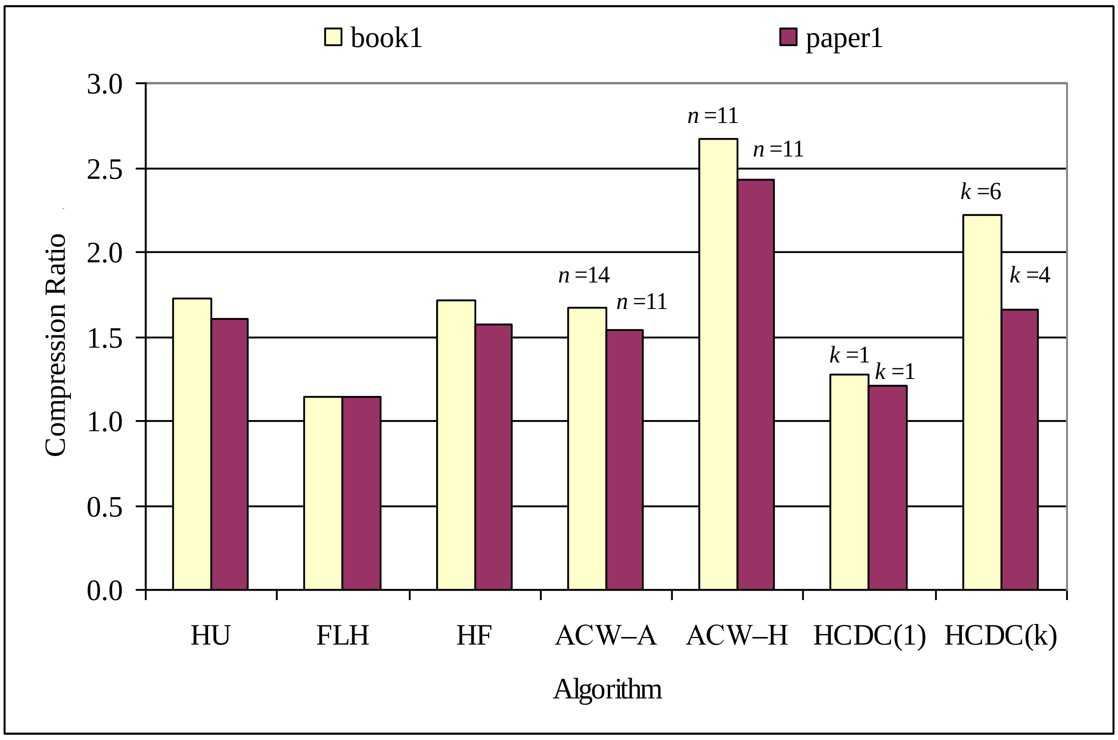


Figure (4.5) - Comparison between the compression ratio of the HCDC(*k*) scheme and various statistical and bit-level data compression algorithms.

4.4. Experiment #4: Comparing the Compression Ratio of the HCDC (k) Scheme with Adaptive Algorithms

Table (4.5) compares the compression ratio of the new scheme with three adaptive algorithms, namely, the Unix compact utility that is based on adaptive Huffman (AH), the greedy adaptive Fano coding (AF) [Rue 06], the ACW(*n*,*s*) scheme using adaptive coding (ACW-A), and the ACW(*n*,*s*) scheme using Huffman coding (ACW-H) [Hay 08]. Once again the HCDC(*k*) achieve a competitive compression ratio with the other algorithms.

The compression ratio of the new scheme is higher than the compression ratio of AH, AF, and the ACW-A algorithms by 40 to 50 percent. But at the same time it is less than the compression ratio achieved by the ACW-H scheme by 4 to 6 percent, this is because the ACW-H scheme itself consists of two compression algorithms, the ACW(*n*,*s*) followed by Huffman coding [Hay 08]. This demonstrates the excellent performance of the HCDC(*k*) scheme as it provides almost the same compression ratio achieved by using two compression algorithms in a successive way. The results are also shown in graphical form in Figure (4.6).

Table (4.5)

Comparison of the compression ratio of HCDC(*k*) scheme with various adaptive data compression algorithms.

Corpus	File Name	AH[1]	AF[1]	ACW (*n*,*s*) scheme[2]		HCDC(*k*)
				Adaptive	Huffman	
Calgary corpus	Bib	1.526	1.524	1.537 (11)	2.330 (11)	1.428 (4)
	book1	1.753	1.750	1.674 (14)	2.673 (11)	2.225 (6)
	book2	1.658	1.653	1.545 (11)	2.530 (11)	1.971 (5)
	paper1	1.587	1.588	1.542 (11)	2.431 (11)	1.658 (4)
Canterbury corpus	alice29	1.753	1.746	1.656 (14)	2.643 (11)	2.097 (5)
	Asyoulik	1.648	1.645	1.648 (14)	2.516 (11)	1.825 (5)
	lcet10	1.718	1.717	1.604 (14)	2.599 (11)	2.009 (5)

	plrabn12	1.769	1.766	1.750 (14)	2.667 (11)	2.255 (6)

AH (Adaptive Huffman): Unix compact utility.

AF (Adaptive Fano): Greedy adaptive Fano coding.

[1] Results for AH , AF are from [Rue 06].

[2] Results for the ACW-A and ACW-H are from [Hay 08].

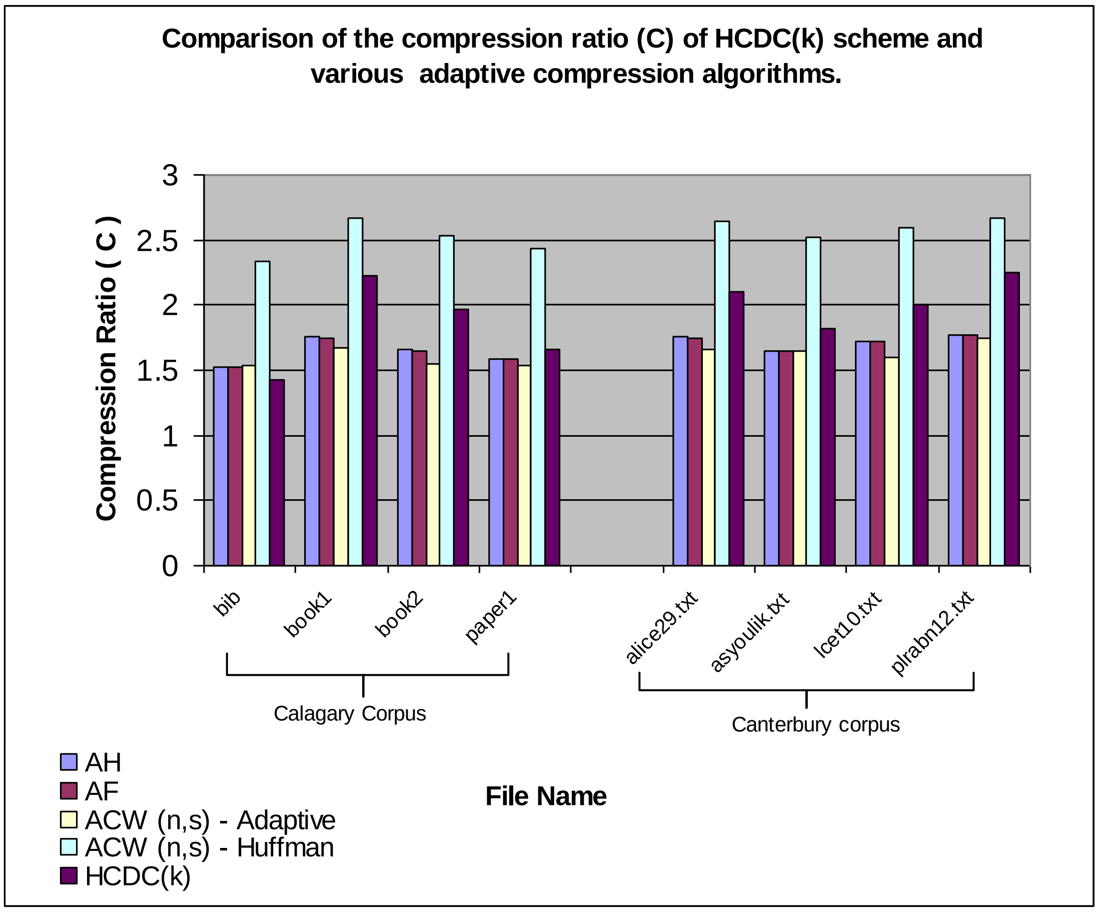


Figure (4.6) - Comparison between the compression ratio of the HCDC(*k*) scheme with various adaptive compression algorithms for a number of text files from the Calgary and Canterbury corpora.

4.5. Experiment #5: Comparing Compression Ratio of the HCDC (k) Scheme with a Number of Widely-Used Programs

In this experiment the performance of the HCDC(*k*) scheme and a number of well-known and widely-used data compression programs, is compared. The performance is compared in terms of the coding rate (C_r) in bpc for six text files. These are: Bib, book1, book2, paper1, paper2, from the Calgary corpus and alice29 from the Canterbury corpus. The coding rates achieved for these text files are tabulated in Table (4.6). The results obtained for book 1 file are also presented in Figures (4.7).

There are no special reasons for selecting these text files and for comparing the performance in terms of coding rate, apart from the fact that such comparison were found in many references in the literature.

It can be easily seen from the results obtained that the coding rates achieved by the HCDC(*k*) scheme with adaptive coding for different text files are competitive to the coding rates achieved by a number of standard programs. This may be considered as an excellent performance as most of these programs utilize a number of data compression algorithms work in successive forms to achieve such coding rates. For example, for the text file book1, the best coding rate achieved is 2.120 bpc by the rkive-mt3 program, which is based on a prediction with partial matching (PPM) approach. However, the HCDC(*k*) scheme achieves a coding rate of 2.753 bpc at *k*=6.

Table (4.6) – Experiment #5
Comparison of the coding rate (C_r) in bpc between the HCDC(*k*) scheme and various standard programs.

Programs	Type	Version	File Name					
			Bib	book1	book2	paper1	paper2	alice29
compress	LZ	4.0	3.350	3.460	3.280	3.770	3.520	-
gzip	LZ	1.2.3	2.520	3.260	2.710	2.800	2.900	-
comp-2-o-4	PPM	Trial V.	2.020	2.350	2.080	2.480	2.450	-
DD	PPM	Trail V.	2.530	2.690	2.390	3.080	2.850	-
compress	LZ	4.3d	-	3.486	-	-	-	3.270
pkzip	LZ	2.04e	-	3.288	-	-	-	2.884
gzip-9	LZ	1.2.4	-	3.250	-	-	-	2.848
szip-b41-o0	BW	1.05Xf	-	2.345	-	-	-	2.239

ha a2	PPM	0.98	-	2.453	-	-	-	2.171
boa-m15	PPM	0.58b	-	2.204	-	-	-	2.061
rkive-mt3	PPM	1.91b1	-	2.120	-	-	-	2.055
neural small	NN	P5	-	2.508	-	-	-	2.301
neural large	NN	P6	-	2.283	-	-	-	2.129
ACW-A[1]	Bit-Level	Trial V.	4.554 (11)	4.182 (14)	4.531 (11)	4.540 (11)	4.266 (14)	4.227 (14)
ACW-H[1]	Bit-Level	Trial V.	3.004 (11)	2.619 (11)	2.767 (11)	2.880 (11)	2.662 (11)	2.649 (11)

HCDC(*k*)	Bit-Level	Trial V.	4.901 (4)	3.146 (6)	3.551 (5)	4.221 (4)	3.213 (6)	3.338 (5)
[1] Results for the ACW-A and ACW-H are from [Hay 08].								

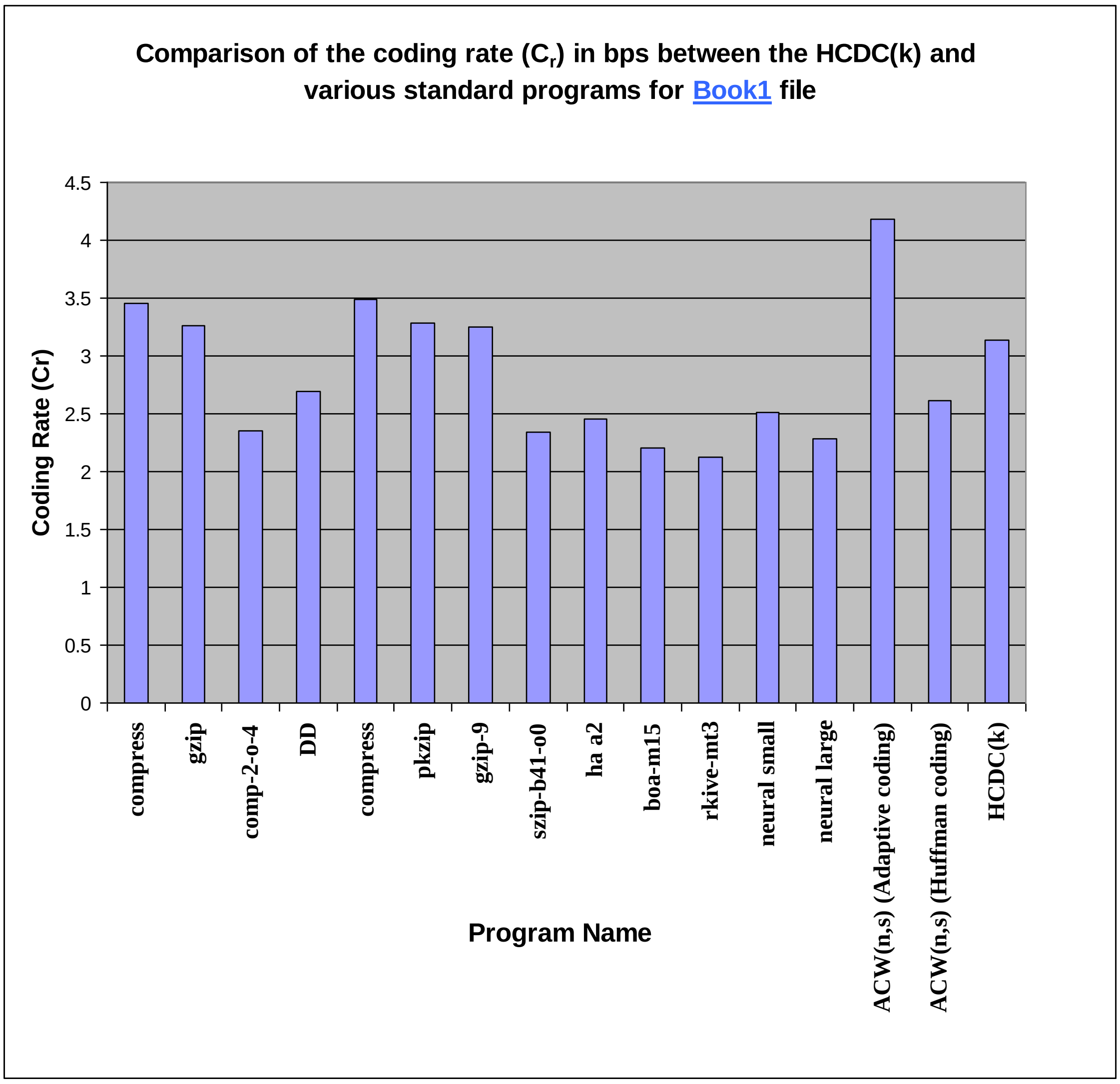


Figure (4.7) -Comparison of the coding rate (C_r) in bpc between the HCDC(*k*) and various standard programs for book1 file from the Calgary corpus.

Chapter Five
Conclusions and Recommendations for Future Work

5.1. Conclusions

The main conclusions of this thesis are:

1. An efficient and adaptive lossless bit-level text compression scheme that is based on the error correcting Hamming Code Data Compression (HCDC) algorithm was developed. The new scheme consists of six main steps, in which steps 2 to 5 are applied repetitively until inflation is detected. Therefore, it is referred to as HCDC(*k*) scheme, where *k* refers to the number of iterations or loops being performed before an inflation is detected.
2. Using adaptive coding instead of ASCII coding in converting a text file into a binary sequence enhances the performance of the HCDC(*k*) scheme. This is because adaptive coding yields low entropy binary sequence so that higher compression ratio is granted. It has been recognized that the entropy of the binary sequence increases as the compression process goes on (i.e., *k* increases) until the entropy reaches or becomes close to unity. Therefore, the compression ratio is either saturated or inflated.

   For example, as demonstrated in Experiment #1, for the text files paper1 and book1 from the Calgary corpus, due to the use of adaptive coding, the compression ratio increases by 34% (from 1.234(2) to 1.656(4)) and by 70% (from 1.309(2) to 2.225(6)), respectively.

However, this is usually at the cost of increasing the processing time, as the values of *k* at which maximum compression ratio is achieved are increased.

3. The repetitive approach implemented in the HCDC(*k*) scheme enhances the compression ratio of the HCDC algorithm by more than 50%, as it has been demonstrated in Experiment #2. However, the maximum value of *k* that gives maximum compression ratio is in the range of 4 to 6 for a wide range of text files from different standard corpora. The results showed that the range of the improvement achieved by the HCDC(*k*) scheme depends on the content and size of the text file.
4. As demonstrated by Experiments #3 and #4, the HCDC(*k*) scheme demonstrated an excellent performance as compared to the most widely used data compression algorithms, such as Huffman coding (HU), the fixed-length Hamming (FLH), and the Huffman coding following the fixed-length Hamming (HF), the

5. Adaptive Character Wordlength (ACW(*n*,*s*)) scheme using adaptive coding (ACW-A), adaptive Huffman (AH), the greedy adaptive Fano coding (AF). However, it provides less compression ratio as compared to the ACW(*n*,*s*) scheme using Huffman coding (ACW-H), because this scheme is in fact two compression algorithms (Huffman and ACW(*n*,*s*)) performed consecutively.

5. Experiment #5 demonstrated that The HCDC(*k*) scheme provides an excellent, comparable, and competitive performance to the most widely used state-of-the-art software of different models.

6. Finally, it is believed that the HCDC(*k*) scheme has a great potential that can be utilized to increase its compression power. In addition, the HCDC(*k*) scheme can be used as a post processing technique to increase the compression ratio of statistical lossless data compression algorithms, such as Shanon-Fano coding, Huffman coding, arithmetic coding, a combination of these algorithms, or any modified form of them.

5.2. Recommendations for Future Work

The main recommendations for future work are:

1. Develop more efficient text-to-binary coding formats to be used with the HCDC(*k*) scheme.
2. Evaluate the performance of the HCDC(*k*) scheme in compressing other types of files, such as standstill images, multimedia files, etc.
3. Use the HCDC(*k*) scheme as a post-compression stage to other data compression algorithms, in particular, bit-level data compression algorithms.
4. Develop an optimized version of the code to compare its runtime with other compression algorithms and state-of-the-art software and compare the compression and the decompression processing runtimes.
5. Modify the core of HCDC algorithm itself in some way to provide higher compression ratio.

References

[Adi 07]	J. Adiego, G. Navarro, and P. de la Fuente, "Using Structural Contexts to Compress Semi-Structured Text Collections", Information Processing and Management, Vol. 43, Issue 3, pp. 769–790, 2007.
[Adi 06]	J. Adiego and P. Feunte, "On the Use of Words as Source Alphabet Symbols in PPM", Proceedings of the IEEE Data Compression Conference (DCC’06), IEEE CS Press, pp. 435-441, 2006.
[Bah 08a]	Hussein Al-Bahadili, "A Novel Lossless Data Compression Scheme Based on the Error Correcting Hamming Codes", Journal of Computers and Mathematics with Applications, Vol. 56, Issue 1, pp. 143-150, 2008.
[Bah 08b]	Hussein Al-Bahadili and Shakir M. Hussain, "An Adaptive Character Wordlength Algorithm for Data Compression", Journal of Computers & Mathematics with Applications, Vol. 55, Issue 6, pp. 1250-1256, 2008.
[Bah 07a]	Hussein Al-Bahadili and Ahmad Rababa’a, “An Adaptive Bit-Level Text Compression Scheme Based on the HCDC Algorithm”, Proceedings of the Musharaka International-Conference on Communications, Networking and Information Technology (MIC-CNIT2007), Amman, Jordan, 6-8 Dec 2007.
[Bel 94]	T. C. Bell and I. Witten, “The Relationship between Greedy Parsing and Symbol Wise Text Compression”, Journal of ACM, Vol. 41, No. 4, 1994.
[Bel 90]	T. C. Bell, J. C. Cleary, and I. H. Witten, "**Text Compression**", Prentice-Hall, 1990.

[Bri 07]	N. J. Brittian, M. R. El-Sakka, "Grayscale True Two-Dimensional Dictionary-Based Image Compression", Journal of Visual Communication and Image Representation, Vol. 18, pp. 35-44, 2007.
[Bri 05]	N. R. Brisaboa, A. Farina, G. Navarro and J. Param´a, "Efficiently Decodable and Searchable Natural Language Adaptive Compression", ACM 1-59593-034, August 2005.
[Cai 04]	G. Caire, S. Shamai, and S. Verdu, “Noiseless Data Compression with Low-Density Parity-Check Codes”, in Discrete Mathematics and Theoretical Computer Science, AMS (2004).
[Chu 02]	A. Chu, “LZAC Lossless Data Compression”, In Proceedings of the Data Compression Conference (DCC’02), IEEE, 2002.
[Con 06]	E. Conley and S. Klein, ”Compression of Multilingual Aligned Texts”, In Proceedings of the Data Compression Conference, IEEE Computer Society, 2006.
[Fre 04]	V. Freschi and A. Balliol, "Longest Common Subsequence Between Run-Length-Encoded String: a New Algorithm with Improved Parallelism", Information Processing letters, Vol. 90, pp. 167-173, 2004.
[Gal 07]	L. Galambos, J. Lansky, M. Zemlicka, and K. Chernik, "Compression of Semi-Structured Documents", International Journal of Information Technology, Vol. 4, No. 1, pp. 11-17, 2007.
[Gal 78]	R. G. Gallager, "Variations on a Theme by Huffman", IEEE Trans. on Info. Theory, Vol. 24, No. 6, pp. 668-674, November 1978.

[Gil 06]	J. Gilbert and D. Abrahamson, "Adaptive Object Code Compression", ACM 1-59, 593-543, October 2006.
[Has 03]	R. Hashemian, "Direct Huffman Coding And Decoding Using The Table Of Code-Lengths", In Proceedings of the International Conference on Information Technology, IEEE, 2003.
[Hay 08]	Wiam Y. Al_Hayek, "Development and Performance Evaluation of a Bit-Level Text Compression Scheme Based on the Adaptive Character Wordlength Algorithm", M.Sc. Thesis, Department of Computer Science, Graduate College of Computing Studies, Amman Arab University for Graduate Studies, Jordan, 2008
[How 94]	P. G. Howard and J. S. Vitter, "Arithmetic Coding for Data Compression", Proceedings of the IEEE, Vol. 82, No. 6, pp. 857-865, 1994.
[Huf 52]	D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of IRE, Vol. 40, No. 9, pp. 1098-1101, 1952.
[Isa 01]	R. Y. K. Isal and A. Moffat, "Word-Based Block-Sorting Text Compression", Proceedings of the 24th Australasian Computer Science Conference, pp. 92–99, 2001.
[Jar 06]	A.. Jaradat, M. Irshid and T. Nassar," A File Splitting Technique for Reducing the entropy of text files", Int. Journal of Information Technology ,Vol. 3, No. 2, 2006.
[Jar 01]	A. Jaradat and M. Irshid, "A Simple Binary Run-Length Compression Technique For Nonbinary Sources Based On Source Mapping", Active and Passive Elec. Comp., Vol. 24, pp. 211-221, 2001.

[Kim 05]	N. Kimura and S.Latifi, "A Survey on Data Compression in Wireless Sensor Networks", Proceedings of the IEEE International Conference on Information Technology: Coding and Computing (ITCC'05), pp. 8-13, 2005.
[Kle 00]	S. T. Klein, "Skeleton Trees for Efficient Decoding of Huffman Encoded Texts", Information Retrieval, Vol. 3, pp. 7–23, 2000.
[Knu 85]	D. E. Knuth, "Dynamic Huffman Coding", Journal of Algorithms, Vol. 6, pp. 163-180, 1985.
[Lan 06b]	J. Lansky and M. Zemlicka, "Compression of Small Text Files Using Syllables", Proceedings of the IEEE Data Compression Conference (DCC'06), IEEE CS Press, pp. 458-464, 2006.
[Lan 05]	J. Lansky and M. Zemlicka, "Text Compression: Syllables", Proceedings of the Dateso 2005 Annual International Workshop on Databases, Texts, Specifications and Objects (eds. K. Richta, V. Snasel, and J. Pokorny), Vol. 129, pp. 32-45, 2005.
[Lel 87]	D. A. Lelewer and D. S. Hirschberg , "Data Compression", ACM Computing Surveys, Vol. 19, No. 3, pp. 261-296, 1987.
[Liu 05]	Y. K. Liu and B. Zalik, "An Efficient Chain Code with Huffman Coding", Pattern Recognition, Vol. 38, pp. 553-557, 2005.
[Mah 00]	M. V. Mahoney, "Fast Text Compression with Neural Networks", Proceedings of the 13th International Florida Artificial Intelligence Research Society Conference, pp. 230-234, 2000.
[Mof 05]	A. Mofat and R. Y. Isal, "Word-Based Text Compression Using the Burrows-Wheeler Transform", Information Processing and Management, Vol. 41, pp. 1175-1192, 2005.

[Mof 98]	A. Moffat, R. M. Neal, I. H. Witten, "Arithmetic Coding Revisited", ACM Transactions on Information Systems, Vol. 16, pp. 256-294, July 1998.
[Nel 89]	M. Nelson , "LZW Data Compression", Dr Dobb's Journal, Vol. 14, No. 10, pp. 62-75, 1989.
[Nof 07]	S. Nofal, "Bit-Level Text Compression", Proceedings of the First International Conference on Digital Communications and Computer Applications, pp. 486-488, 2007.
[Pan 00]	M. K. Pandya, "Data Compression: Efficiency of Varied Compression Techniques", Formal Report, University of Brunel, UK, 2000.
[Pla 06]	H. Plantinga, "An Asymmetric, Semi-Adaptive Text Compression Algorithm", IEEE Data Compression, 2006.
[Rei 06a]	S. Rein, C. Guhmann, F. Fitzek, "Compression of Short Text on Embedded Systems", Journal of Computers, Vol. 1, No. 6, 2006.
[Rei 06b]	S. Rein, C. G¨uhmann and F. Fitzek, "Low Complexity Compression of Short Messages", Proceedings of the IEEE Data Compression Conference (DCC'06), IEEE CS Press, pp. 123-132, 2006.
[Rob 06]	L. Robert and R. Nadarajan, "New Algorithms for Random Access Text Compression", Proceedings of the international conference, IEEE, pp. 104-111, 2006.
[Rue 01]	L. Rueda and b. John, "Enhanced Static Fano Coding", In the Int. Conference, IEEE, Vol. 4, pp. 2163–2169, 2001.
[Sal 04]	D. Salomon, **Data Compression: The Complete Reference**, Third Edition, Springer-Verlag, 2004.

[Sha 06]	D. Shapira and A. Daptardar, "Adapting the Knuth–Morris–Pratt Algorithm for Pattern Matching in Huffman Encoded Texts", Information Processing and Management, Vol. 42, pp. 429–439, 2006.
[Sha 04]	A. A. Sharieh, "An Enhancement of Huffman Coding for the Compression of Multimedia Files ", Transactions of Engineering Computing and Technology, Vol. 3, No. 1, pp. 303-305, 2004.
[Sha 51]	C. E. Shannon," Prediction and Entropy of Printed English", The Bell System Technical Journal, 1951.
[Tan 03]	Amdrew Tanenbaum, "Computer Networks", Prentice Hall, 2003.
[Vit 89]	J. S. Vitter , "Dynamic Huffman Coding", Journal of ACM, Vol. 15, No. 2, pp. 158-167, 1989.
[Wit 04]	I. H. Witten, ""Adaptive Text Mining: Inferring Structure from Sequences", Journal of Discrete Algorithms, Vol. 2, No. 2, pp. 137-159, 2004.
[Wit 94]	I. H. Witten, A. Moffat, and T. Bell, "Managing Gigabytes: Compressing and Indexing Documents and Images", Van Nostrand Reinhold, 1994.
[Wit 87]	I. H. Witten, R. M. Neal , and J. C. Cleary , "Arithmetic Coding for Data Compression", Communications of the ACM, Computing Practice, Vol. 30, No. 6, pp. 520-540, 1987.
[Xie 03]	Y. Xie, W. Wolfe, and H. Lekatsas, "Code Compression Using Variable-to-Fixed Coding Based on Arithmetic Coding", In Proceedings of the Data Compression Conference, IEEE Computer Society, 2003.
[Ziv 78]	J. ZIV and A. Lempel, "Compression of Individual Sequence Via Variable-Rate Coding", IEEE Transaction on Information Theory, Vol. 4, No. 5, pp. 530-536, 1978.

[Ziv 77]	J. ZIV and A. Lempel ,"A Universal Algorithm for Sequential Data Compression", IEEE Transaction on Information Theory, Vol. 23, No. 3, pp. 337-343, 1977.

Appendices

Appendix A
Compression Corpora

In order to evaluate the performance of various compression schemes, standard corpora are usually used, these include:

i. Calgary Corpus

ii. Canterbury Corpus

iii. Artificial Corpus

iv. Large Corpus

v. Miscellaneous Corpus

This appendix provides brief descriptions of the above corpora and their constituent files.

A.1 Calgary Corpus

The Calgary Corpus is the most referenced corpus in the data compression field, especially, for text compression and is the de facto standard for lossless compression evaluation. The corpus was founded in 1987 by Ian Witten, Timothy Bell and John Cleary [Wit 87, Bel 89, Bel 90]. There are two versions of this corpus:

1. Large Calgary corpus which consists of 18 files (bib, book1, book2, geo, news, obj1, obj2, paper1, paper2, paper3, paper4, paper5, paper6, pic, progc, progl, progp and trans).
2. Standard Calgary Corpus which consists of 14 files (all files above except paper3, paper4, paper5 and paper6).

Nine different types of text are represented, and to confirm that the performance of schemes is consistent for any given type, many of the types have more than one representative. Normal English, both fiction and non-fiction, is represented by two books and six papers (labeled book1, book2, paper1, paper2, paper3, paper4, paper5, paper6). More unusual styles of English writing are found in a bibliography (bib) and a batch of unedited news articles (news).

Three computer programs represent artificial languages (progc, progl, progp). A transcript of a terminal session (trans) is included to indicate the increase in speed that could be achieved by applying compression to a slow line to a terminal. All of the files mentioned so far use ASCII encoding. Some non-ASCII files are also included: two files of executable code (obj1, obj2), some geophysical data (geo), and a bit-map black and white picture (pic).

The file geo is particularly difficult to compress because it contains a wide range of data values, while the file pic is highly compressible because of large amounts of white space in the picture, represented by long runs of zeros. More details of the individual texts are given in [Bel 90]. In addition, results of compression experiments on these texts are given in [Wit 87, Bel 89].

Table (A.1) - Calgary Corpus.

#	File Name	Size (Byte)	Contents
1	Bib	111261	Structured text (bibliography)
2	Book1	768771	Text
3	Book2	610856	Formatted text, scientific
4	Geo	102400	Geophysical data
5	News	377109	Formatted text, script with news
6	Obj1	21504	Program code (object file), executable machine code
7	Obj2	246814	Program code (object file), executable machine code
8	Paper1	53161	Formatted text, scientific
9	Paper2	82199	Formatted text, scientific
10	Paper3	46526	Formatted text, scientific
11	Paper4	13286	Formatted text, scientific
12	Paper5	11954	Formatted text, scientific
13	Paper6	38105	Formatted text, scientific
14	Pic	513216	Image data (black and white)
15	Progc	39611	Source code
16	Progl	716 46	Source code
17	Progp	493 79	Source code

18	Trans	93695	Transcript terminal data

A.2 Canterbury Corpus

The Canterbury collection is the main benchmark for comparing compression methods. It was developed in 1997 as an improved version of the Calgary Corpus. It consists of 11 files. The files were chosen because their results on existing compression algorithms are "typical", and so it is hoped this will also be true for new methods. Ross Arnold and Tim Bell [Arn 97] explain how the files were chosen, and why it is difficult to find "typical" files. This collection will not be changed so that it can be used as a benchmark in future.

Table (A.2) - Canterbury Corpus.

#	File Name	Size (Byte)	Contents
1	Alice29.txt	152089	English text
2	Asyoulik.txt	125179	Shakespeare
3	Cp.html	24603	HTML source
4	Fields.c	11150	C source
5	Grammar.lsp	3721	LISP source
6	Kennedy.xls	1029744	Excel Spreadsheet
7	Lcet10.txt	426754	Technical writing

8	Plrabn12.txt	481861	Poetry
9	Ptt5	513216	CCITT test set
10	Sum	38240	SPARC Executable
11	Xargs.1	4227	GNU manual page

A.3 Artificial Corpus

The Artificial corpus is a collection that contains 4 files for which the compression methods may exhibit pathological or worst-case behavior - files containing little or no repetition (random.txt), files containing large amounts of repetition (alphabet.txt), or very small files (a.txt).

As such, “average” results for this collection will have little or no relevance, as the data files have been designed to detect outliers. Similarly, times for “trivial” files will be negligible, and should not be reported.

New files can be added to this collection, so the overall average for the collection should not be reported as a benchmark. Results on this corpus should be reported for individual files, or a subset should be identified. Existing files in the collection will not be changed or removed.

Table (A.3) - Artificial Corpus.			
#	File Name	Size (Byte)	Contents
1	a.txt	1	The letter “a”.
2	aaa.txt	100000	The letter “a”, repeated 100,000 times.
3	alphabet.txt	100000	Enough repetitions of the alphabet to fill 100,000 characters
4	random.txt	100000	100,000 characters, randomly selected from [a-z, A-Z, 0-9] (alphabet size 64)

A.4 Large Corpus

The Large Corpus is a collection of relatively 3 large files. While most compression methods can be evaluated satisfactorily on smaller files, some require very large amounts of data to get good compression, and some are so fast that the larger size makes speed measurement more reliable. New files can be added to this collection, so the overall average for the collection should not be reported as a benchmark. Results on this corpus should be reported for individual files, or a subset should be identified. Existing files in the collection will not be changed or removed.

Table (A.4) - Large Corpus.			
#	File Name	Size (Byte)	Contents
1	E.coli	4638690	Complete genome of the E. Coli bacterium (E.coli).
2	bible.txt	4047392	The King James version of the bible (bible).
3	world192.txt	2473400	The CIA world fact book (world).

A.5 Miscellaneous Corpus

This is a collection of "miscellaneous" files that is designed to be added to by researchers and others wishing to publish compression results using their own files. New files can be added to this collection, so the overall average for the collection should not be reported as a benchmark.

Results on this corpus should be reported for individual files, or a subset should be identified. Existing files in the collection will not be changed or removed. There is only one file in this corpus till now.

Table (A.5) - The Miscellaneous Corpus.			
#	File Name	Size (Byte)	Contents
1	pi.txt	10000000	The first million digits of pi.